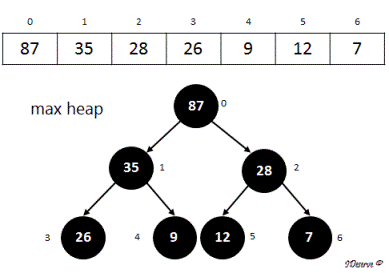
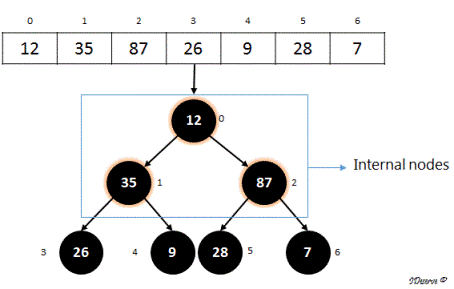
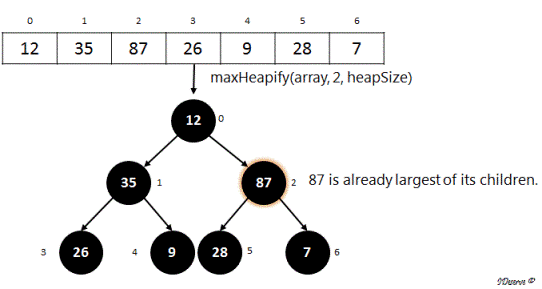
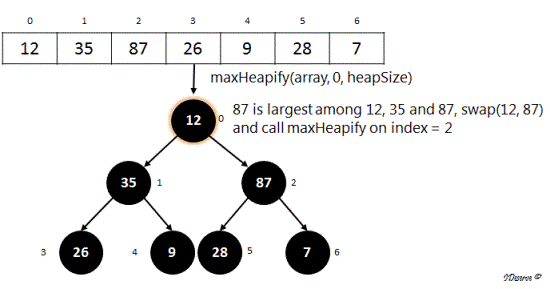
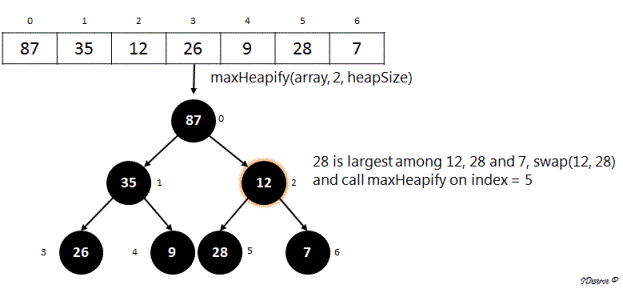
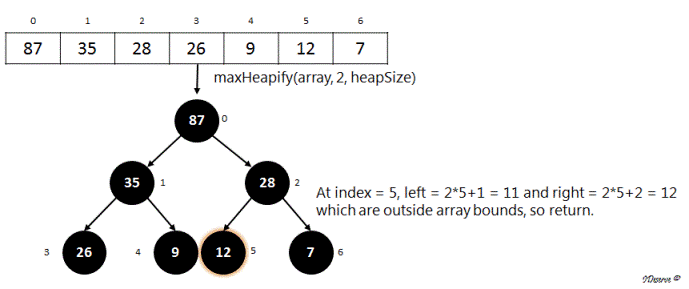
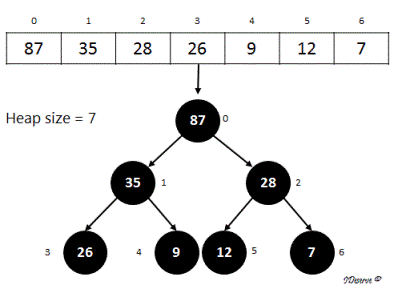
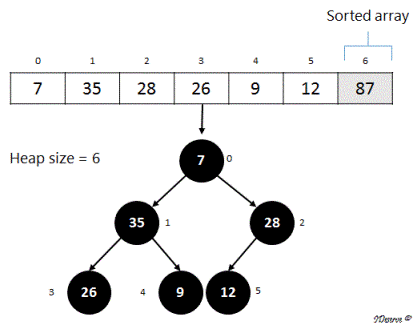
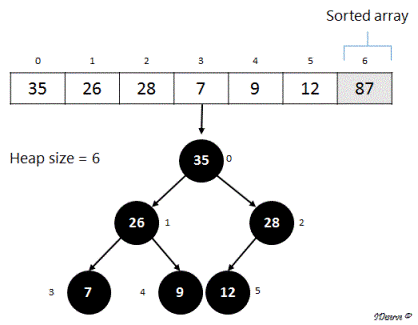
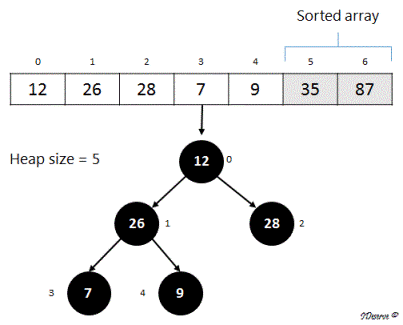
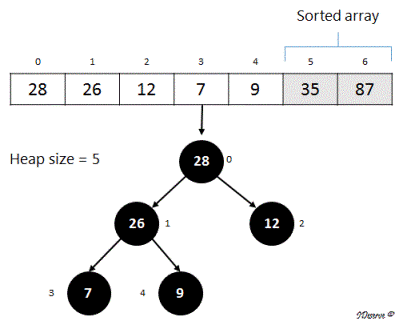
Sorting Algorithm - Heap Sort

## Algorithm/Insights

Heap sort is an [in-place](https://en.wikipedia.org/wiki/In-place_algorithm) [comparison sort](https://en.wikipedia.org/wiki/Comparison_sort).  
Heap sort is **not** a [stable sort](https://en.wikipedia.org/wiki/Stable_sort) i.e. it does not ensure the order of same array elements after sorting.  
  
Definition:  
A max-heap is a complete binary tree in which the value at root is greater than or equal to the values of left and right children of the root and all the heap nodes follow this property.  
Since a max heap is a complete binary tree, it can be represented by an array such that:  
1. Element at index 0 is the root.  
2. For any node of the heap at index i, including the root node (i=0):   
   Index of Left child  = 2\*i+1  
   Index of Right child = 2\*i+2  
   Parent of the node   = (i-1)/2  
  
  
Heap sort algorithm has two steps:  
**Step 1. Build Max Heap: Convert the array to a max heap.**  
Call maxHeapify on all internal nodes. For this we start with the last internal node upto the root. The index of the last internal node will be the index of parent of last leaf node which is also the last element in the array.  
maxHeapify(array, i, heapSize) - Get larger of current node (element at index i), left child (element at index 2\*i+1) and   
right child (element at index 2\*i+2) and replace it with the current node (index i).   
Recurse for left and right nodes to maintain max heap property.  
  
**Step 2: Sorting the array using Max Heap**  
1. Remove the maximum element from the heap (root) and swap it with the last index of the heap (heapSize) and reduce the heap size by 1.   
2. Then adjust heap elements (from index 0 to heap size) to maintain max heap property.  
3. Repeat this step for second largest element and so on.  
  
For Example, consider the array:  
12, 35, 87, 26, 9, 28, 7  
  
**Step 1: Build Max Heap:**  
1. The array can be considered as following complete binary tree:  
  
2. Clearly,it is not a max heap. We convert it to a max heap by calling maxHeapify on all internal nodes.  
Last internal node  
    = index of parent of last element of the array  
    = index of parent of 7         (Index of 7 is 6)  
    = (6 - 1)/2  
    = 2  
    = index of 87  
  
There will be no change in the array on calling maxHeapify(array, 2, heapSize).  
Similarly, subtree with root as 35 is also the largest of its children, i.e. satisfies max heap property.  
So, there will be no change in the array on calling maxHeapify(array, 1, heapSize) too.  
Next we call, maxHeapify(array, 0, heapSize):  
  
  
  
Hence following image shows the max heap created from the array.  
  
  
**Step 2: Sorting the array using Max Heap created in Step 1**  
1. Swap max element of the heap (i.e. the root) with the element at **last index of the heap**.  
  
2. Call maxHeapify on the root (index = 0) to maintain max heap property on the remaining heap. Proceeding like before, we get the following heap:  
  
3. Next swap max element of the heap (root) with the element at last index of the heap.  
  
4. Again, call maxHeapify on the root (index = 0) to maintain max heap property on the remaining heap.  
  
  
In the next step, 28 (root) will be swapped with 9 (last element in heap) moving 28 to its correct position in the sorted array.  
Continuing like this, we can see that the array is getting sorted from max element to smallest element using the max heap.  
After applying #1 and #2 of step 2 on the array till heapSize becomes 0, we will get the sorted array as:  
[7, 9, 12, 26, 28, 35, 87]

|  |
| --- |
| import java.util.Arrays;    /\*\*   \* <b>IDeserve <br>   \* <a href="<https://www.youtube.com/c/IDeserve>">[https://www.youtube.com/c/IDeserve</a](https://www.youtube.com/c/IDeserve%3C/a)>   \* Given an array, sort the array using Heap Sort algorithm.   \*   \* @author Saurabh   \*/  public class HeapSort {        public static void maxHeapify(int[] array, int curIndex, int heapSize){          // Left child in heap          int left = 2\*curIndex+1;          // Right child in heap          int right = 2\*curIndex+2;          int largest = curIndex;            if(left < heapSize && array[left] > array[curIndex]) {              largest = left;          }            if(right < heapSize && array[right] > array[largest]) {              largest = right;          }            if(largest != curIndex){              swap(array, curIndex, largest);              maxHeapify(array, largest, heapSize);          }      }        public static void buildMaxHeap(int[] array, int heapSize){          // call maxHeapify on all internal nodes          int lastElementIndex = array.length - 1;          int parentIndex = (lastElementIndex - 1)/2;          for(int i = parentIndex; i >= 0; i--){              maxHeapify(array, i, heapSize);          }      }        public static void heapSort(int[] array){          if(array == null || array.length < 2)              return;            buildMaxHeap(array, array.length);          int heapSize = array.length;          for(int i = array.length - 1; i > 0; i--){              swap(array, 0, i);              heapSize = heapSize - 1;              maxHeapify(array, 0, heapSize);          }      }        public static void main(String[] args) {          int[] array = {12, 35, 87, 26, 9, 28, 7};          System.out.println("Original Array:\t\t" + Arrays.toString(array));          heapSort(array);          System.out.println("Sorted Array:\t\t" + Arrays.toString(array));      }        private static void swap(int[] array, int i, int j) {          int tmp = array[i];          array[i] = array[j];          array[j] = tmp;      }  } |

## Order of the Algorithm

Time Complexity is O(nlogn)  
Space Complexity is O(1)

# Rotate an Array

Given an array and a positive integer k, rotate the array k times.  
Example:  
Array:  1 2 3 4 5  
k: 1  
Output: 2 3 4 5 1  
k: 2  
Output: 3 4 5 1 2  
k: 3  
Output: 4 5 1 2 3

## Video coming soon!

#### Subscribe for more updates

## Algorithm/Insights

Naive Solution: Rotate array by one element k times  
Algorithm to rotate array by one element:  
1. Take a temporary variable to store first element of the array.  
2. Shift all elements by 1 position on the left, overwriting the first element of the array.  
3. Set last element of array to the first element saved in temporary variable.  
This is rotation by 1 element.  
By invoking this k times, we get array rotated k times.  
Time Complexity: O(n\*k)  
Space Complexity: O(1)  
  
Solution using Temporary Array:   
1. Move first k elements of the array to a temporary array.  
2. Shift all the elements of the array from k+1 to end of array to the beginning of the array.  
3. Copy the k elements in temporary array to the last k positions of the original array.  
Time Complexity: O(n)  
Space Complexity: O(k)  
  
Efficient Solution: Using Array Reversal  
1. Reverse the array elements from 0 to k-1.  
2. Reverse the array elements from k to array.length-1.  
3. Reverse the whole array.  
Example:   
1 2 3 4 5  
k = 3  
Step 1. Reverse the array elements from 0 to 2: 3 2 1 4 5  
Step 2. Reverse the array elements from 3 to 4: 3 2 1 5 4  
Step 3. Reverse the whole array: 4 5 1 2 3  
Time Complexity: O(n)  
Space Complexity: O(1)  
  
What happens if k > n?  
Example:   
array: 1 2 3 4 5  
k = 8  
Then after 8 rotations, the array will be:  
4 5 1 2 3  
which is same as rotating the array 3 times (k%n = 8%5 = 3)  
If this is not handled in algorithms 2 (using temporary array) and 3 (using array reversal), we get an ArrayIndexOutOfBoundsException.   
So, we add following checks on k in all 3 algorithms:  
1. If k < 0, throw IllegalArgumentException  
2. If k > n, set k = k%n

import java.util.Arrays;

/\*\*

 \* <b>IDeserve <br>

 \* <a href="<https://www.youtube.com/c/IDeserve>">[https://www.youtube.com/c/IDeserve</a](https://www.youtube.com/c/IDeserve%3C/a)>

 \* Given an array and a positive integer k, rotate the array k times.

 \* Example:

 \* Array:  1 2 3 4 5

 \* k: 3

 \* Output: 4 5 1 2 3

 \*

 \* @author Saurabh

 \*/

public class RotateArray {

    // Rotate array by using reversing operation on the array

    // O(n) time, O(1) space

    public static void rotateArrayUsingReverse(int[] array, int k) {

        if(k < 0) {

            throw new IllegalArgumentException("k cannot be negative!");

        }

        if(array == null || array.length < 2) {

            return;

        }

        int n = array.length;

        if(k > n)

            k = k%n;

        reverseArray(array, 0, k-1);

        reverseArray(array, k, n-1);

        reverseArray(array, 0, n-1);

    }

    // Reverse an array between start (s) and end (e)

    private static void reverseArray(int[] array, int s, int e) {

        while(s < e) {

            int tmp = array[s];

            array[s] = array[e];

            array[e] = tmp;

            s++;

            e--;

        }

    }

    // Solution using temporary array

    // O(n) time, O(k) space

    public static void rotateArrayUsingTmp(int[] array, int k) {

        if(k < 0) {

            throw new IllegalArgumentException("k cannot be negative!");

        }

        if(array == null || array.length < 2) {

            return;

        }

        int n = array.length;

        if(k > n)

            k = k%n;

        int[] tmp = new int[k];

        for(int i = 0; i < k; i++) {

            tmp[i] = array[i];

        }

        for(int i = k; i < n; i++) {

            array[i-k] = array[i];

        }

        for(int i = n - k, j = 0; i < n; i++, j++) {

            array[i] = tmp[j];

        }

    }

    // Naive Solution - O(n\*k)

    // Rotate array by using the method rotateArrayOnce k times

    public static void rotateArrayNaive(int[] array, int k) {

        if(k < 0) {

            throw new IllegalArgumentException("k cannot be negative!");

        }

        if(array == null || array.length < 2) {

            return;

        }

        int n = array.length;

        if(k > n)

            k = k%n;

        for(int i = 0; i < k; i++) {

            rotateArrayOnce(array);

        }

    }

    // Rotate array 1 element at a time

    public static void rotateArrayOnce(int[] array) {

        int first = array[0];

        for(int i = 0; i < array.length-1; i++) {

            array[i] = array[i+1];

        }

        array[array.length-1] = first;

    }

    public static void main(String[] args) {

        testArrayRotationNaive();

        testArrayRotationTmp();

        testArrayRotationReverse();

    }

    private static void testArrayRotationNaive() {

        int[] array = {1,2,3,4,5};

        int k = 8;

        System.out.println("Original Array: ");

        System.out.println(Arrays.toString(array));

        rotateArrayNaive(array, k);

        System.out.println("After Rotation " + k + " times using naive algorithm: ");

        System.out.println(Arrays.toString(array));

    }

    private static void testArrayRotationTmp() {

        int[] array = {1,2,3,4,5};

        int k = 8;

        System.out.println("Original Array: ");

        System.out.println(Arrays.toString(array));

        rotateArrayUsingTmp(array, k);

        System.out.println("After Rotation " + k + " times using temporary array: ");

        System.out.println(Arrays.toString(array));

    }

    private static void testArrayRotationReverse() {

        int[] array = {1,2,3,4,5};

        int k = 8;

        System.out.println("Original Array: ");

        System.out.println(Arrays.toString(array));

        rotateArrayUsingReverse(array, k);

        System.out.println("After Rotation " + k + " times using reversal: ");

        System.out.println(Arrays.toString(array));

}}

# Fibonacci Number

In mathematics, the Fibonacci series is defined by the following recurrence relation:  
F(0) = 0  
F(1) = 1  
F(n) = F(n-1) + F(n-2)  
i.e. nth element is formed by adding elements at (n-1) and (n-2)  
So, first 10 fibonacci numbers will be:  
0, 1, 1, 2, 3, 5, 8, 13, 21, 34  
  
Given a number n, find F(n).  
Example:   
Input: 6  
Output: 8  
Input: 10  
Output: 55

## Video coming soon!

#### Subscribe for more updates

## Algorithm/Insights

Recursive Solution:  
1. Boundary Condition: If n = 0 or n = 1, return n  
2. Return F(n-1) + F(n-2)  
F(n-1) = F(n-2) + F(n-3)  
We see that F(n-2) will be computed again for F(n-1). So, a lot of redundant calculations.  
Time Complexity: Exponential  
  
Efficient Solution:  
The problem can be solved by Dynamic Programming because it has both the properties:  
1. Overlapping Subproblems -   
        As F(n) = F(n-1) + F(n-2), computation of nth fibonacci number requires (n-1) and (n-2) fibonacci numbers.   
        Further F(n-1) = F(n-2) + F(n-3), and so on  
        So, we see that there are overlapping sub problems.  
2. Optimal Substructure -  
        If subproblems F(n-1) and F(n-2) are computed optimally, then the solution to F(n) is also optimal. Hence it satisfies optimal substructure property.  
So, if we store the solutions to subproblems here, we can find next fibonacci number efficiently without computing the values again.  
Also, as F(n) = F(n-1) + F(n-2), we need only the last 2 values of the series to find the next value. Hence, we do not need to store all subproblem solutions but only the previous 2 i.e. F(i-1) and F(i-2). Once we have F(i), for calculating F(i+1) we just need F(i) and F(i-1), and therefore we can discard F(i-2) as it is not needed further.  
So starting with F(0) = 0, F(1) = 1, we can use bottom up approach to calculate F(n)  
1. Boundary Condition: If n = 0 or n = 1, return n  
2. Take a = 0, b = 1  
3. Loop n-2 times, and calculate  
   c = a+b  
   a = b  
   b = c  
4. Return c

public class Fibonacci {

    // Recursive Solution

    // Time complexity: Exponential

    public static int getFibonacciRec(int n) {

        if(n < 0) {

            throw new IllegalArgumentException("n cannot be negative!");

        }

        if(n == 0 || n == 1)

            return n;

        return getFibonacciRec(n-1) + getFibonacciRec(n-2);

    }

    public static int getFibonacci(int n) {

        if(n < 0) {

            throw new IllegalArgumentException("n cannot be negative!");

        }

        if(n == 0 || n == 1)

            return n;

        int a = 0, b = 1;

        int c = a+b;

        for(int i = 2; i <= n; i++) {

            c = a+b;

            a = b;

            b = c;

        }

        return c;

    }

    public static void main(String[] args) {

        //System.out.println(getFibonacciRec(6));

        System.out.println(getFibonacci(6));

    }

}

# Merge two sorted arrays without using extra space

Given two sorted arrayA and arrayB such that arrayA has enough void spaces in it to be able to accommodate arrayB in it. Void spaces in an array are denoted using INVALID\_NUM. Write a program to merge arrayB into arrayA such that resulting array is a sorted array. The expected space complexity is O(1).    
  
For example, if arrayA = {-3, 5, INVALID\_NUM, 7, INVALID\_NUM, 10, INVALID\_NUM, 11, INVALID\_NUM} and arrayB = {-1, 2, 6, 12}  
then arrayS should be modified to array - {-3, -1, 2, 5, 6, 7, 10, 11, 12}

## Algorithm/Insights

The algorithm to solve this problem is very similar to merge procedure of mergesort algorithm. First we move all valid elements of arrayA towards the end of it. For example, arrayA = {-3, 5, INVALID\_NUM, 7, INVALID\_NUM, 10, INVALID\_NUM, 11, INVALID\_NUM} is modified to {INVALID\_NUM,INVALID\_NUM,INVALID\_NUM,INVALID\_NUM,-3,5,7,10,11}. After this modification, k is initialized to 0, element arrayA[i] is compared with element arrayB[j] for values of 'i' starting from index of first valid element in arrayA up to size of arrayA and for all values of 'j' starting from 0 up to size of arrayB. If element arrayA[i] is less than element arrayB[j], arrayA[i] is copied to arrayA[k] otherwise arrayB[j] is copied to arrayA[k]. This compare and copy operation is done until all the elements from either arrayA or arrayB are compared and copied to arrayA. If after completion of this operation, there are still elements left in arrayB to be copied in arrayA then remaining elements from arrayB are copied towards the end of the arrayA.  
  
Please checkout function inplaceMergeArrays(int[] arrayA, int[] arrayB) in code snippet for implementation details. The time complexity of this algorithm is O(n) with space complexity of O(1).

|  |
| --- |
| public class MergeSortedArraysInplace  {      final static int INVALID\_NUM = 0;        public void inplaceMergeArrays(int[] arrayA, int[] arrayB)      {          // move all elements of arrayA with valid values towards the end          int validNumIndex = arrayA.length - 1;          for (int i = arrayA.length - 1; i >= 0; i--)          {              if (arrayA[i] != INVALID\_NUM)              {                  arrayA[validNumIndex] = arrayA[i];                  validNumIndex -= 1;              }          }            // i: index of first valid valued element in arrayA          int i = validNumIndex + 1;          int j = 0, k = 0;            // fill-up arrayA starting from 0th index since this end of arrayA is free now          while ((i < arrayA.length) && (j < arrayB.length))          {              if (arrayA[i] < arrayB[j])              {                  arrayA[k++] = arrayA[i++];              }              else              {                  arrayA[k++] = arrayB[j++];              }          }            // copy any remaining elements of smaller array into larger one          while (j < arrayB.length)          {              arrayA[k++] = arrayB[j++];          }      }        public static void main(String[] args)      {          MergeSortedArraysInplace solution = new MergeSortedArraysInplace();              int[] arrayA = {-3, 5, INVALID\_NUM, 7, INVALID\_NUM, 10, INVALID\_NUM, 11, INVALID\_NUM};          int[] arrayB = {-1, 2, 6, 12};            solution.inplaceMergeArrays(arrayA, arrayB);          for (int i = 0;  i < arrayA.length; i++)          {              System.out.print(arrayA[i] + ", ");          }      }  } |

## Order of the Algorithm

Time Complexity is O(n)  
Space Complexity is O(1)

# Maximum subarray sum

Given an array of unordered positive and negative integers, find the maximum subarray sum in the array.  
For example:  
Array: {2, -9, 5, 1, -4, 6, 0, -7, 8}  
Output:  
Maximum subarray sum is 9

## Algorithm/Insights

1. Start with curSum = 0 and maxSum = 0.  
2. Traverse the array elements, from i = 0 to n-1, where n is the length of the array.  
   a. Keep adding array elements to curSum till curSum >= 0 and update maxSum to curSum if maxSum < curSum.  
   b. As soon as curSum becomes < 0, reset curSum = 0.  
3. Finally return maxSum.  
This is known as Kadane's algorithm.  
  
The above method does not take care of the case when all the elements in the array are negative.  
In this case, we can modify the algorithm, to keep a flag hasAllNegativeNumbers initialized to true.  
If a positive or 0 element is seen in the array, then set hasAllNegativeNumbers = false.  
Also, while traversing the array, keep track of maximum negative element.  
Finally, if hasAllNegativeNumbers is true, then return maxNegativeSum, else return maxSum.

public class MaxSubarraySum {

    public static int findMaxSubarraySum(int[] input) {

        int curSum = 0;

        int maxSum = 0;

        boolean hasAllNegativeNumbers = true;

        int maxNegativeSum = Integer.MIN\_VALUE;

        for(int i = 0; i < input.length; i++) {

            if(hasAllNegativeNumbers && input[i] >= 0) {

                hasAllNegativeNumbers = false;

            } else if(hasAllNegativeNumbers && input[i] < 0 && maxNegativeSum < input[i]) {

                maxNegativeSum = input[i];

            }

            curSum += input[i];

            if(curSum < 0) {

                curSum = 0;

            }

            if(maxSum < curSum) {

                maxSum = curSum;

            }

        }

        return hasAllNegativeNumbers ? maxNegativeSum : maxSum;

    }

    public static void main(String[] args) {

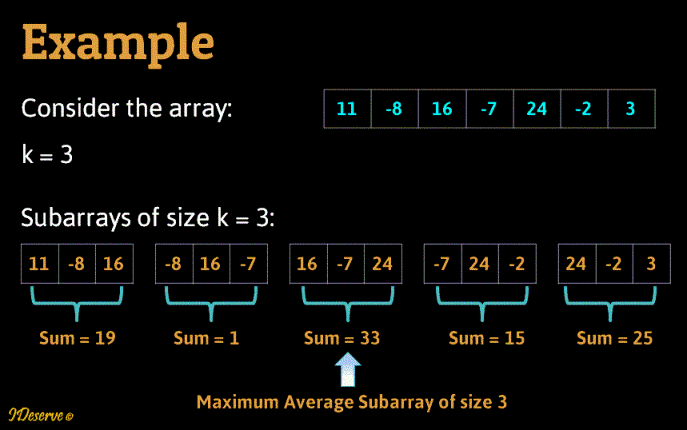
        int[] input = {2, -9, 5, 1, -4, 6, 0, -7, 8};

        System.out.println("Maximum subarray sum is " + findMaxSubarraySum(input));

    }

}

# Maximum average subarray of size k

Given an integer array and a number k, print the maximum sum subarray of size k.  
Maximum average subarray of size k is a subarray (sequence of contiguous elements in the array) for which the average is maximum among all subarrays of size k in the array.  
Average of k elements = (sum of k elements)/k  
Since k > 0, the maximum sum subarray of size k will also be maximum average subarray of size k. So, the problem reduces to finding maximum sum subarray of size k in the array.  


We use sliding window strategy for solving this.  
Step 1: Find sum of first k elements in the input array. Initialize maxSum to the calculated sum and maxSumStartIndex = 0.  
Step 2: Add next element to the sum and subtract first element from the sum. Check if this sum is greater than previous sum and update maxSum and maxSumStartIndex.  
Step 3: Keep adding next element to the sum and removing first element from the sum to get sum of current sub array of size k and update maxSum and maxSumStartIndex whenever a greater sum is seen.  
Step 4: Finally print k elements starting from maxSumStartIndex.

public class MaxAvgSubarray {

    private static int getMaxAvgSubarrayStartIndex(int input[], int k)

    {

        int n = input.length;

        if (k > n)

            throw new IllegalArgumentException("k should be less than or equal to n");

        if(k == n) {

            return 0;   // whole array is the solution

        }

        // Sum of first k elements

        int sum = input[0];

        for (int i = 1; i < k; i++)

            sum += input[i];

        // Initialized to first k elements sum

        int maxSum = sum;

        int maxSumIndex = 0;

        // Sum of remaining sub arrays

        for (int i = k; i < n; i++)

        {

            // Remove first element of the current window and add next element to the window

            sum = sum - input[i-k] + input[i] ;

            if (sum > maxSum)

            {

                maxSum = sum;

                maxSumIndex = i-k;

            }

        }

        // Return starting index of max avg sub array

        return maxSumIndex + 1;

    }

    public static void printMaxAvgSubarray(int[] input, int k) {

        System.out.print("Maximum average subarray of length " + k + " is:  ");

        int index = getMaxAvgSubarrayStartIndex(input, k);

        for(int i =0 ; i < k; i++) {

            System.out.print(input[index++] + " ");

        }

    }

    public static void main(String[] args) {

        int[] input = {11, -8, 16, -7, 24, -2, 3};

        int k = 3;

        printMaxAvgSubarray(input, k);

}}

## Order of the Algorithm

Time Complexity is O(n)  
Space Complexity is O(1)

# Longest Substring with non-Repeating Characters

Given a string, find the longest substring with non-repeating characters.  
Example:  
Input : ABCDABDEFGCABD  
Output: ABDEFGC

## Algorithm/Insights

Method 1: Naive Algorithm  
Check for every substring whether it is has repeating characters. If not, then check if it is longest substring or not.  
Time complexity: O(n^3)  
Space Complexity: O(1)  
  
Method 2: Linear time Algorithm  
1. Create an array charIndexes which has last index of string characters seen in the string, initialized to -1  
2. Traverse the array and check if current character was seen earlier in the current sub-array, if not seen then increment index of current non repeating substring seen till now(currLength).   
3. If the current character is a repeated character and length of longest substring (maxLength) seen till now is less than current length, then update maxLength.  
Time complexity: O(n)  
Space Complexity: O(1)

public class LongestSubstring {

    // Naive algorithm for finding the longest substring without repeating characters

    // Quadratic time complexity

    public static String getLongestSubstringNonRepeatingCharsNaive(String str) {

        String longestSubstring = "";

        for(int i = 0; i < str.length(); i++) {

            for(int j = i; j < str.length(); j++) {

                if(hasRepeatingChars(str, i, j)) {

                    break;

                } else if(longestSubstring.length() < j-i+1){

                    longestSubstring = str.substring(i, j+1);

                }

            }

        }

        return longestSubstring;

    }

    private static boolean hasRepeatingChars(String str, int start, int end) {

        boolean[] isCharPresent = new boolean[256];

        for(int i = 0; i < 256; i++) {

            isCharPresent[i] = false;

        }

        for(int i = start; i <= end; i++) {

            if(isCharPresent[str.charAt(i)]) {

                return true;

            } else {

                isCharPresent[str.charAt(i)] = true;

            }

        }

        return false;

    }

    // Linear time algorithm for finding the longest substring without repeating characters

    public static String getLongestSubstringNonRepeatingChars(String str) {

        if(str == null) {

            return null;

        }

        int n = str.length();

        if(n < 2) {

            return str;

        }

        // array to store last index of string characters seen in the string, initialized to -1

        int[] charIndexes = new int[256];

        for(int i = 0; i < n; i++) {

            charIndexes[i] = -1;

        }

        // Set index of first character

        charIndexes[str.charAt(0)] = 0;

        int currLength = 1; // Length of current non repeating substring

        int maxLength = 1;  // Length of longest substring with non repeating characters found

        int prevIdx = 0;    // previous index of current character

        int startIdx = 0;   // Starting index of longest substring with non repeating characters found

        for(int i = 1; i < n; i++) {

            prevIdx = charIndexes[str.charAt(i)];

            if(prevIdx == -1 || i - currLength > prevIdx) {

                currLength++;

            } else {

                if(currLength > maxLength) {

                    maxLength = currLength;

                    startIdx = i - maxLength;

                }

                currLength = i - prevIdx;

            }

            charIndexes[str.charAt(i)] = i;

        }

        // Check if longest substring with non repeating characters ends at end of the string

        if(currLength > maxLength) {

            maxLength = currLength;

            startIdx = n - maxLength;

        }

        return str.substring(startIdx, startIdx + maxLength);

    }

    public static void main(String[] args) {

        String str = "ABCDABDEFGCABD";

        /\*String longestSubstring = getLongestSubstringNonRepeatingCharsNaive(str);

        System.out.println("Longest substring non repeating chars by Naive method:\t\t" + longestSubstring);\*/

        String longestSubstring = getLongestSubstringNonRepeatingChars(str);

        System.out.println("Longest substring non repeating chars by linear time method:\t" + longestSubstring);

    }

}

# Leaders in an array

Given an array of integers, print the leaders in the array. A leader is an element which is larger than all the elements in the array to its right.  
For example:  
Input Array:  
{ 98, 23, 54, 12, 20, 7, 27 }  
Output:  
Leaders- 27 54 98

## Algorithm/Insights

Start from the right keeping track of largest element (currentLeader). If a larger element is found, then print it and set currentLeader to the current element.  
Note that the last element is a leader since there is no element to its right.  
The solution can be visualized by considering array elements as towers and printing towers which are visible if seen through right view of the array.  
Please note that leaders will be seen in sorted order when seen from right(right view of the towers).

public class LeaderElements {

    public static void printLeaders(int[] input) {

        if(input == null || input.length == 0) {

            return;

        }

        int inputSize = input.length;

        int currentLeader = input[inputSize - 1];

        System.out.print("Leaders- ");

        for (int i = inputSize - 1; i >= 0; i--) {

            if(input[i] >= currentLeader) {

                currentLeader = input[i];

                System.out.print(currentLeader + " ");

            }

        }

    }

    public static void main(String[] args) {

        int[] input = { 98, 23, 54, 12, 20, 7, 27 };

        printLeaders(input);

    }

}

# Find the missing number in the duplicate array

Given two integer arrays where second array is duplicate of first array with just 1 element missing. Find the element.  
Example:   
  
Input:  
Array1 - 9 7 8 5 4 6 2 3 1  
Array2 - 2 4 3 9 1 8 5 6  
  
Output:  
7

## Algorithm/Insights

Algorithm 1:  
1. Iterate over array1, find sum of all numbers of array1, say result.  
2. Iterate over array2 and subtract every element of array2 from result.  
3. Print result.  
Typically in an interview, there can be a follow-up question to solve this problem without using arithmetic operators. Following is the algorithm for the same.  
  
Algorithm 2:  
1. Initialize result = 0.  
2. Iterate over both the input arrays and XOR 'result' with each element of the input arrays.  
3. Print result.  
Please check out the video above for detailed explanation of the algorithm with animations.

|  |
| --- |
| public class MissingNumberInDuplicateArray {        /\*       \* Find missing number using xor operation.       \* Handled for the generic case where any of the input array could be       \* duplicate of the other one.       \*/      public static void missingNumberInDuplicateArray(int[] array1, int[] array2) {          if(array1 == null && array2 == null) {              System.out.println("Input arrays are empty!");          } else if(array1 == null) {              if(array2.length == 1) {                  System.out.println("Missing element = " + array2[0]);              } else {                  System.out.println("Input is not valid!");              }          } else if(array2 == null) {              if(array1.length == 1) {                  System.out.println("Missing element = " + array1[0]);              } else {                  System.out.println("Input is not valid!");              }          } else {              int len1 = array1.length;              int len2 = array2.length;              if(Math.abs(len1 - len2) != 1) {                  System.out.println("Input is not valid!");                  return;              }              if(len1 > len2){                  findMissingNumber(array1, array2);              } else {                  findMissingNumber(array2, array1);              }          }      }        private static void findMissingNumber(int[] array1, int[] array2) {          int result = array1[0];          for (int i = 1; i < array1.length ; i++) {              result = result ^ array1[i];          }          for (int i = 0; i < array2.length ; i++) {              result = result ^ array2[i];          }          System.out.println("Missing element = " + result);      }        /\*       \* Find missing number using sum of array elements.       \* Handled for the generic case where any of the input array could be       \* duplicate of the other one.       \*/      public static void missingNumberInDuplicateArrayUsingSum(int[] array1, int[] array2) {          if(array1 == null && array2 == null) {              System.out.println("Input arrays are empty!");          } else if(array1 == null) {              if(array2.length == 1) {                  System.out.println("Missing element = " + array2[0]);              } else {                  System.out.println("Input is not valid!");              }          } else if(array2 == null) {              if(array1.length == 1) {                  System.out.println("Missing element = " + array1[0]);              } else {                  System.out.println("Input is not valid!");              }          } else {              int len1 = array1.length;              int len2 = array2.length;              if(Math.abs(len1 - len2) != 1) {                  System.out.println("Input is not valid!");                  return;              }              if(len1 > len2){                  findMissingNumberUsingSum(array1, array2);              } else {                  findMissingNumberUsingSum(array2, array1);              }          }      }        private static void findMissingNumberUsingSum(int[] array1, int[] array2) {          int result = 0;          for(int i = 0; i < array1.length; i++) {              result += array1[i];          }          for(int i = 0; i < array2.length; i++) {              result -= array2[i];          }          System.out.println("Missing element = " + result);      }        public static void main(String[] args) {          int[] array1 = {9, 7, 8, 5, 4, 6, 2, 3, 1};          int[] array2 = {2, 3, 4, 9, 1, 8, 5, 6};          missingNumberInDuplicateArray(array1, array2);      }  } |

## Order of the Algorithm

Time Complexity is O(n)  
Space Complexity is O(1)

# Find Minimum Length Sub Array With Sum K

Given an array A having positive and negative integers and a number k, find the minimum length sub array of A with sum = k.

## Algorithm/Insights

1. Iterate over the array using 2 loops.   
2. Initialize currentSum = 0, min = MAX\_VALUE  
3. Starting from array[i], keep adding array[i] to currentSum till currentSum != k or till last element of the array or size of current subarray becomes > min.  
4. If currentSum == k update min.  
5. Also keep track of start and end index of the min subarray obtained so far.  
6. Print array elements from start to end.

public class MinimumLengthSubArraySum {

    public static void main(String[] args) {

        int[] array = {2,3,1,2,4,3};

        int k = 7;

        printMinSubArrayWithSum(array, k);

    }

    public static void printMinSubArrayWithSum(int[] array, int k) {

        int start = -1;     // Start of min subarray

        int end = -1;       // End of min subarray

        int min = Integer.MAX\_VALUE;    // size of the smallest subarray with sum = k

        for(int i = 0; i < array.length; i++) {

            int currentSum = 0;

            for(int j = i; j < array.length && (j-i+1) < min; j++) {

                currentSum += array[j];

                if(currentSum == k) {

                    start = i;

                    end = j;

                    min = end - start + 1;

                    break;

                }

            }

        }

        if(start == -1 || end == -1)  {

            System.out.println("No subarray exists with sum = " + k);

            return ;

        }

        System.out.print("[ ");

        while(start <= end) {

            System.out.print(array[start] + " ");

            start++;;

        }

        System.out.println("]");

    }

Time Complexity is O(n^2)  
Space Complexity is O(1)

# Binary Search in a Sorted Array

## Algorithm/Insights

Binary search algorithm is used for solving many programming interview problems.   
This algorithm is based on divide and conquer strategy to find a number in a sorted integer array.  
Binary Search:  
1. Take start = 0, end = length of array - 1.  
2. Repeat following steps till start <= end:    
   a). Set mid = (start + end)/2.  
   b). Check if array[mid] == num, then return mid.  
   c). If num < array[mid], set end = mid - 1  
   d). Else set start = mid+1  
3. Return -1.

# First non-repeating character in a string

Return the first such character found.  
  
Linear Solution 1: Use a count array to store the count of every character.  
1. Create a count array.  
2. Initialize count of every element to 0.  
3. Traverse the string once and increment the count of every element found in the string.  
4. Traverse the string again and check for the first element for which count is set as 1. Return this element.  
5. If not found, return null  
If string is large, traversing string again to find first element whose count is 1, is inefficient.  
  
Linear Solution 2 (Optimized): Use index array for storing the index of the string elements.  
1. Create an index array.  
2. Initialize the index of all index array elements to -1.  
3. Traverse the string once and for element of the string, check the value of index of that string element in index array.  
   a. If index is -1, it is the first occurrence in the string. Set index[string.charAt(i)] = i  
   b. Else, the element is repeating, set index[string.charAt(i)] = -2  
4. Traverse the index array once, find the minimum value in the array which is non negative. If found, this is the index of the first non repeating character in the string.  
   Else, return null.

|  |
| --- |
| public class FirstNonRepeatingCharacter {        public static Character getFirstNonRepeatingCharacterLinearOptimized(String string) {          if(string == null || string.length() == 0) {              return null;          }            int n = string.length();          if(n == 1) {              return string.charAt(0);          }            int[] charIdx = new int[256];   // Index of non repeating characters. If repeating, then index = -2          // Initialize character index of all characters to -1          for(int i = 0; i < 256; i++) {              charIdx[i] = -1;          }            for(int i = 0; i < n; i++) {              if(charIdx[string.charAt(i)] == -1) {                  // character seen first time                  charIdx[string.charAt(i)] = i;              } else {                  // Repeated character                  charIdx[string.charAt(i)] = -2;              }          }            int minIdx = n; // Index of first non repeating character          for(int i = 0; i < 256; i++) {              if(charIdx[i] >= 0 &&                      minIdx > charIdx[i]) {                  minIdx = charIdx[i];              }          }          return (minIdx >= 0 && minIdx < n) ? string.charAt(minIdx) : null;      }        public static Character getFirstNonRepeatingCharacterLinear(String string) {          if(string == null || string.length() == 0) {              return null;          }            int n = string.length();          if(n == 1) {              return string.charAt(0);          }            int[] charCounts = new int[256];          // Initialize character counts of all characters to 0          for(int i = 0; i < 256; i++) {              charCounts[i] = 0;          }            for(int i = 0; i < n; i++) {              charCounts[string.charAt(i)]++;          }            for(int i = 0; i < n; i++) {              if(charCounts[string.charAt(i)] == 1) {                  return string.charAt(i);              }          }          return null;      }        public static Character getFirstNonRepeatingCharacterNaive(String string) {          if(string == null || string.length() == 0) {              return null;          }            int n = string.length();          for(int i = 0; i < n; i++) {              boolean flag = true;              for(int j = 0; j < n; j++) {                  if(i != j && string.charAt(i) == string.charAt(j)) {                      flag = false;                      break;                  }              }              if(flag) {                  return string.charAt(i);              }          }          return null;      }        public static void main(String[] args) {          String string = "ADBCGHIEFKJLADTVDERFSWVGHQWCNOPENSMSJWIERTFB";          System.out.println("Output: " + getFirstNonRepeatingCharacterLinearOptimized(string));      }  } |

## Order of the Algorithm

Time Complexity is O(n)  
Space Complexity is O(1)

# Re-arrange elements in an array to put positive and negative elements in alternate order

Given an array with positive and negative elements in it, re-arrange these elements such that positive and negative elements are placed in alternate order. Positive elements should be placed at even indices and negative elements should be placed at odd indices. The order of same signed elements should remain same. It is not guaranteed that positive and negative elements are equal in number. If there are more number of positive elements then after arranging positive and negative elements in correct order, remaining positive elements should be placed at the end of the array. Same is the case when there are more number of negative elements. No extra space other than auxiliary variables should be used.  
  
For example, if the input array is {-1,3,2,4,5,-6,7,-9} output should be {3,-1,2,-6,4,-9,5,7}. Note how the order of elements 3,2,4,5,7 and of elements -1,-6,-9 is unchanged among themselves. Also the extra positive elements 5 and 7 are placed at the end of the array. In the sub-array which maintains alternate order, positive elements are placed at even indices and negative elements are placed at odd indices.

## Algorithm/Insights

Before looking at the algorithm, let's first define the right-rotate rotation operation on a given array. If the given array is {-1,3,4,5} then the right-rotate operation on this array would result in array {5,-1,3,4}. Here each element except the last element is right-shifted by one position and the last element is placed at the first position.  
  
Now let's look at the algorithm for re-arranging elements. The steps of the algorithm are -   
1. Traverse the array from index 0 to last index.  
2. If at any 'index', we find that the element a[index] is not at its right position(if positive element is not at even index and vice versa), then we find out the next element say at index 'nextOpposite' which is of opposite sign to a[index]. Once we find this 'nextOpposite' index we do a right-rotate operation on sub-array ['index'  to 'nextOpposite'] including both extremes. The right-rotate operation puts the element a['nextOpposite'] at position 'index' and maintains the order of remaining elements. If we do not find any opposite signed element, then there are extra elements remaining of either positive / negative sign and we break out of the loop since the algorithm is complete.   
  
Let's look at an example to understand this algorithm.  
Input: {-1,3,2,4,5,-6,7,-9}  
1. array = {-1,3,2,4,5,-6,7,-9}, index = 0: -1 is not at its right position, next element of opposite sign(3) is found at index 1, right-rotate for sub-array{-1,3} is done and modified array is {3,-1,2,4,5,-6,7,-9}  
2. array = {3,-1,2,4,5,-6,7,-9}, index = 1: -1 is now placed at its right position.   
3. array = {3,-1,2,4,5,-6,7,-9}, index = 2: 2 is also at its right position.  
4. array = {3,-1,2,4,5,-6,7,-9}, index = 3: 4 is not at its right position, next opposite signed element(-6) is at index 5. Right-rotate operation is performed on sub-array{4,5,-6} which results in modified array {3,-1,2,-6,4,5,7,-9}   
5. array = {3,-1,2,-6,4,5,7,-9}, index = 4: 4 is now placed at its right position.  
6. array = {3,-1,2,-6,4,5,7,-9}, index = 5: 5 is not placed at its right position, right-rotation is performed on sub-array {5,7,-9} which results in modified array {3,-1,2,-6,4,-9,5,7}.  
7. All remaining elements are at their right positions.

public class RearrangeElements

{

    private void leftShift(int[] array, int low, int high)

    {

        int lastElement = array[high];

        for (int i = high; i > low; i--)

        {

            array[i] = array[i-1];

        }

        array[low] = lastElement;

    }

    private boolean notAtRightPosition(int[] array, int index)

    {

        // even indices should have positive elements in them

        if ((index % 2) == 0)

        {

            if ((array[index]) < 0) // if negative element

            {

                return true;

            }

            else

            {

                return false;

            }

        }

        else // odd indices should have negative elements in them

        {

            if ((array[index]) < 0)

            {

                return false;

            }

            else

            {

                return true;

            }

        }

    }

    private int getNextOpposite(int[] array, int index)

    {

        for (int i = index+1; i < array.length; i++)

        {

            if ((array[i]\*array[index]) < 0)

            {

                return i;

            }

        }

        return -1;

    }

    public void reArrangePositivesNegatives(int[] array)

    {

        for (int i = 0; i < array.length; i++)

        {

            if (notAtRightPosition(array, i))

            {

                int nextOppositeIndex = getNextOpposite(array, i);

                if (nextOppositeIndex != -1)

                {

                    leftShift(array, i, nextOppositeIndex);

                }

                else // no more opposite signed elements

                {

                    break;

                }

            }

        }

    }

    public static void main(String[] args)

    {

        RearrangeElements solution = new RearrangeElements();

        int[] testArray = {-1,3,2,4,5,-6,7,-9};

        solution.reArrangePositivesNegatives(testArray);

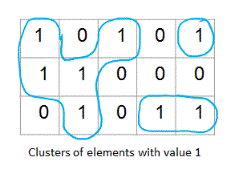
        for(int i = 0;  i < testArray.length; i++)

        {

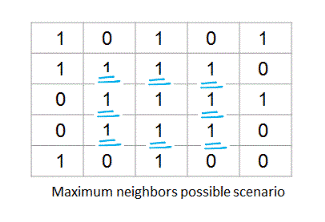
            System.out.println(testArray[i]);

        }

    }

Given a 2D matrix of 0s and 1s, find total number of clusters formed by elements with value 1.  For example, in the below shown 2D matrix there are total three such clusters.   
  
This problem is also known as 'Number of Islands' problem.

## Algorithm/Insights

The algorithm to find total number of clusters of elements with value 1 in a given 2D matrix treats the given matrix as a graph and then it finds out total number of connected components in that graph.  
  
While modeling the 'matrix' as a graph -   
1. An element matrix[i][j] with value 1 is considered as a vertex.  
2. All adjacent elements of matrix[i][j] with value 1 are considered as its neighbor vertices. An element can have maximum of eight neighbors as shown below.  
  
  
With this graph modeling in place, we use following algorithm to find total number of clusters -   
1. Initialize count to 0. Initialize a 2D 'visited' array of booleans which is of size equal to given matrix. Initialize all elements of 'visited' array to false.  
2. For an element matrix[i][j], if matrix[i][j] is '1' and visited[i][j] is 'false' then  
2a. Increment count by 1.  
2b. Start depth first search from element matrix[i][j]. Along with element matrix[i][j], this depth first search would mark all the vertices which are directly or indirectly connected to element matrix[i][j] as visited. In short all the vertices in the cluster starting from vertex matrix[i][j] are visited in this depth first search.  
3. Repeat step #2 for all the elements of given 2D matrix.   
4. Return the 'count' which is basically total number of clusters of 1s in given 2D matrix.  
  
Time complexity of this algorithm is O(n) where is 'n' is total number of elements in the given 2D array. This algorithm uses O(n) extra space to keep track of visited vertices.

public class NumberOfClusters

{

    final static int[] offsets = {-1, 0, +1};

    private boolean neighborExists(int[][] matrix, int i, int j)

    {

        if ((i >= 0) && (i < matrix.length) && (j >= 0) && (j < matrix[0].length))

        {

            if (matrix[i][j] == 1)

            {

                return true;

            }

        }

        return false;

    }

    private void doDFS(int[][] matrix, int i, int j, boolean[][] visited)

    {

        if (visited[i][j])

        {

            return;

        }

        // mark this vertex as visited and visit all its neighbors in depth first manner

        visited[i][j] = true;

        int xOffset, yOffset;

        for (int l = 0; l < offsets.length; l++)

        {

            xOffset = offsets[l];

            for (int m = 0; m < offsets.length; m++)

            {

                yOffset = offsets[m];

                // do not consider vertex[i][j] as its own neighbor

                if (xOffset == 0 && yOffset == 0)

                {

                    continue;

                }

                // check if there exists a vertex at this offset and check if it is '1'

                if (neighborExists(matrix, i + xOffset, j + yOffset))

                {

                    doDFS(matrix, i + xOffset, j + yOffset, visited);

                }

            }

        }

    }

    public int findNumberofClusters(int[][] matrix)

    {

        // JVM initializes all values to false by default.

        boolean[][] visited = new boolean[matrix.length][matrix[0].length];

        int count = 0;

        for (int i = 0; i < matrix.length; i++)

        {

            for (int j = 0; j < matrix[0].length; j++)

            {

                if ((matrix[i][j] == 1) && (!visited[i][j]))

                {

                    // vertex [i][j] marks start of new a cluster. DFS then visits all vertices of this cluster

                    count += 1;

                    doDFS(matrix, i, j, visited);

                }

            }

        }

        return count;

    }

    public static void main(String[] args)

    {

        int[][] matrix = {

                            {1, 0, 1, 0, 1},

                            {1, 1, 0, 0, 0},

                            {0, 1, 0, 1, 1},

                         };

        NumberOfClusters solution = new NumberOfClusters();

        System.out.println(solution.findNumberofClusters(matrix));

    }

# Find the next greater number using same digits

Given a number, find the next greater number using same digits(or by rearranging the digits). For example, if the given number is 1234 then next greater number would be 1243. For the input 1243, next greater number would be 1324. If the input is 6938652 then the output should be the number 6952368.  
  
If there is no next greater number possible, then the program should return the same number. For example, for number 4321, same number 4321 would be returned.

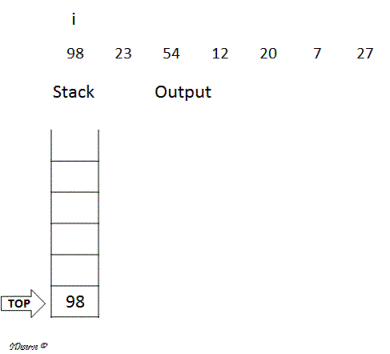
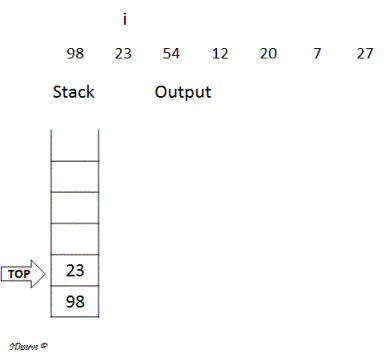
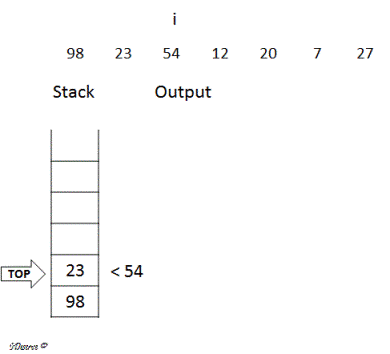
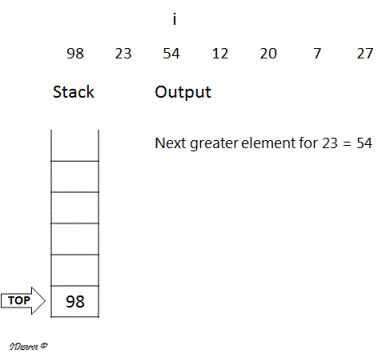
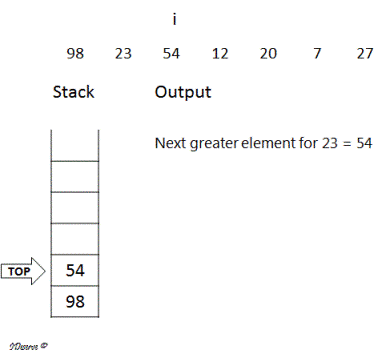
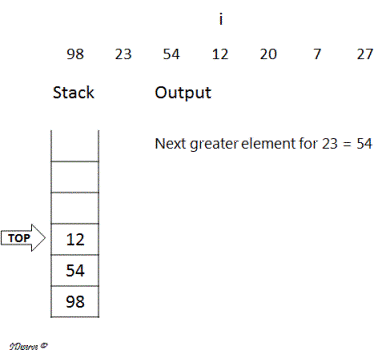
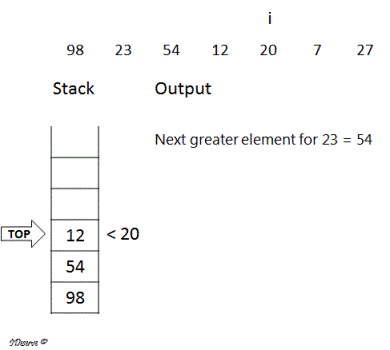
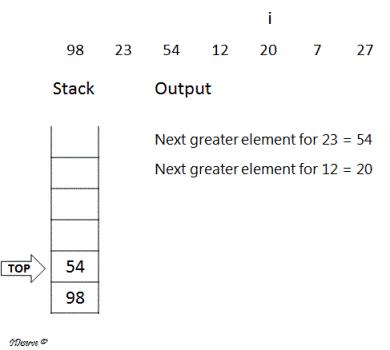
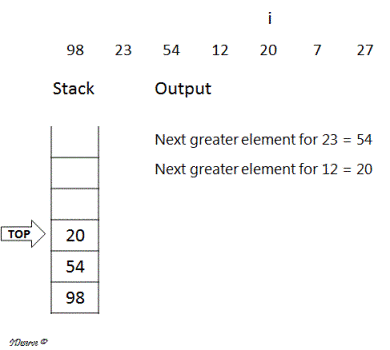
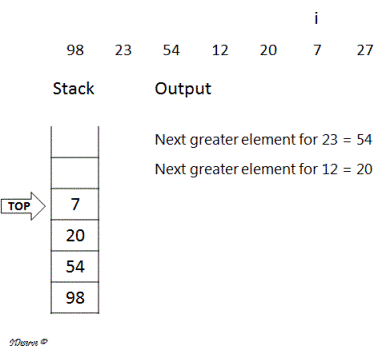
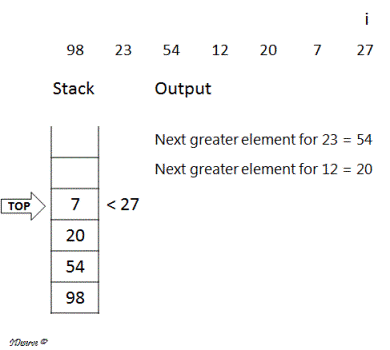
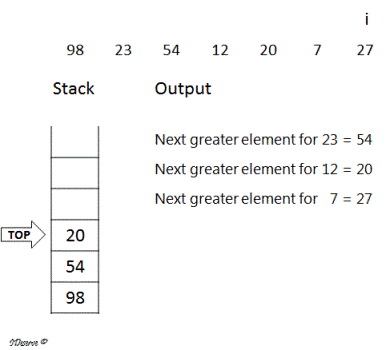
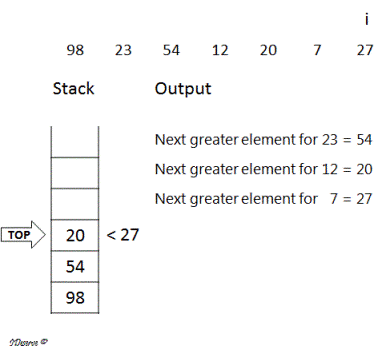
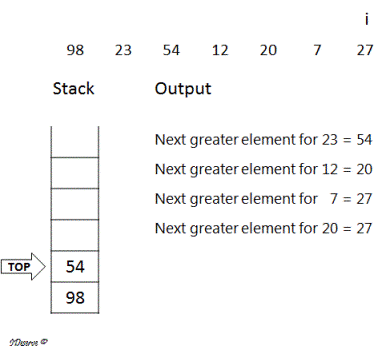
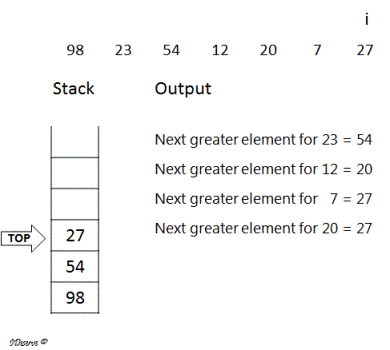
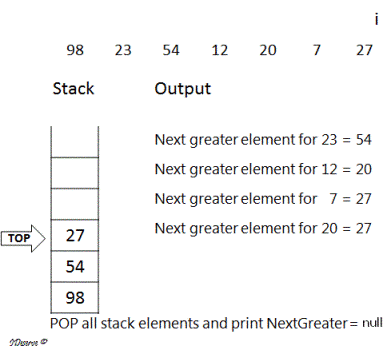
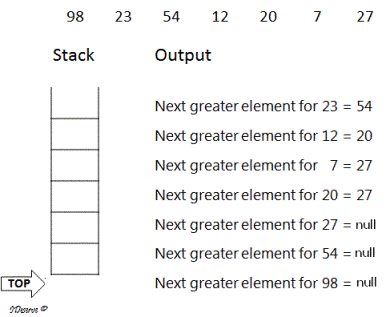
## Algorithm/Insights

We think that the algorithm to solve this problem is best understood using examples. Say we want to get the next higher number for number 1234. If we swap last two digits, we have number 1243 which is next higher number of 1234. Now the next higher number for number 1243 would be 1324. If you observe carefully for input 1243, what we have done is swapped digits 2 and 3 which gets us number - 1342 and then sorted digits after digit 3(42).   
  
Looking at an another example, if the given number is 6938652 then output would be 6952368. Here we have used two steps -   
1. Swap digit 3 and 5 which gives us number 6958632.  
2. Sort digits after the digit 5(8632) which gets us to number 6952368.  
  
Now let us see how we can find out the digits to be swapped in the first step above. We start processing the given number digit by digit from the last digit of the number. In this case(6938652), we start processing from digit 2. We keep looking for the first digit which breaks the sorted ordering of digits. Now observe that the next digit of digit 2 is 5 which is greater than 2 and hence maintains the sorted ordering. Next digit of 5 is 6 which also maintains sorted ordering. Next digit 8 also does not violate sorted ordering. When we look at next digit 3, because it it less than the previously seen digit(8), it is the first digit to break the sorted ordering. Now we find out the next higher digit of digit 3 in the right portion of digit 3 in the number. Next higher digit of digit 3 in portion 8652 is digit 5. Now we swap these two digits which we have found - first digit which breaks the sorted ordering and second digit which is next higher digit in the right portion of first found digit. Once this swap operation is complete we get the intermediate number as 6958632. Now we need to sort the right portion of the index which we found while searching for digit breaking the sorted ordering. We found the digit 3 at index 2, and hence we need to sort right portion of this intermediate number 6958632 starting at index 3 - that portion would be number 8632. After sorting this we get the output as 6952368. As you can confirm, the portion that needs to be sorted would always be in reverse sorted order and therefore reversing this portion would get it in sorted order.  
  
Hopefully, above example has helped to understand the algorithm and its intuition. Now let's look at the formal steps of the algorithm -   
1. Starting from last digit of given number, find the first digit which breaks the sorted ordering. Let the index of this found digit be 'i' and the digit be number[i].  
2. Find the next greater digit in the right portion of number[i] - that is from digit at index i+1 to last digit. Let that digit be number[j] at index 'j'.  
3. Swap digits at index 'i' and 'j'.  
4. Sort the number from index i+1 to end index. Since this portion would be in reverse sorted order, all we need to do is reverse this portion which takes O(n) time.  
  
Please checkout function findNextGreaterNumber(int[] number) for implementation details. Time complexity of this algorithm is O(n) with O(1) extra space.

# Next greater element in an array

Given an array of integers(positive or negative), print the next greater element of all elements in the array. If there is no greater element then print null.  
Next greater element of an array element array[i], is an integer array[j], such that  
1. array[i] < array[j]  
2. i < j  
3. j - i is minimum  
i.e. array[j] is the first element on the right of array[i] which is greater than array[i].  
For example:  
Input array:  98, 23, 54, 12, 20, 7, 27  
Output:  
Next greater element for 23     = 54  
Next greater element for 12     = 20  
Next greater element for 7     = 27  
Next greater element for 20     = 27  
Next greater element for 27     = null  
Next greater element for 54     = null  
Next greater element for 98     = null

## Algorithm/Insights

This problem can be solved by using a stack.  
We traverse the array once.  
1. If the stack is empty or the current element is smaller than top element of the stack, then push the current element on the stack.  
2. If the current element is greater than top element of the stack, then this is the next greater element of the top element. Keep poping elements from the stack till a larger element than the current element is found on the stack or till the stack becomes empty. Push the current element on the stack.  
3. Repeat steps 1 and 2 till the end of array is reached.  
4. Finally pop remaining elements from the stack and print null for them.  
Please note that at any instance, the stack will always be in sorted order having least element at the top and largest element at the bottom.  
  
The algorithm is explained with the help of an example below:  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  


# First non-repeating character in a string

Find the first non repeating character in string.  
Example:  
Input : ADBCGHIEFKJLADTVDERFSWVGHQWCNOPENSMSJWIERTFB  
Output: K

## Algorithm/Insights

Solution 1:  
Naive Solution: Use 2 loops to check for every character whether it is non-repeating or not. Return the first such character found.  
  
Linear Solution 1: Use a count array to store the count of every character.  
1. Create a count array.  
2. Initialize count of every element to 0.  
3. Traverse the string once and increment the count of every element found in the string.  
4. Traverse the string again and check for the first element for which count is set as 1. Return this element.  
5. If not found, return null  
If string is large, traversing string again to find first element whose count is 1, is inefficient.  
  
Linear Solution 2 (Optimized): Use index array for storing the index of the string elements.  
1. Create an index array.  
2. Initialize the index of all index array elements to -1.  
3. Traverse the string once and for element of the string, check the value of index of that string element in index array.  
   a. If index is -1, it is the first occurrence in the string. Set index[string.charAt(i)] = i  
   b. Else, the element is repeating, set index[string.charAt(i)] = -2  
4. Traverse the index array once, find the minimum value in the array which is non negative. If found, this is the index of the first non repeating character in the string.  
   Else, return null.

|  |
| --- |
| public class FirstNonRepeatingCharacter {        public static Character getFirstNonRepeatingCharacterLinearOptimized(String string) {          if(string == null || string.length() == 0) {              return null;          }            int n = string.length();          if(n == 1) {              return string.charAt(0);          }            int[] charIdx = new int[256];   // Index of non repeating characters. If repeating, then index = -2          // Initialize character index of all characters to -1          for(int i = 0; i < 256; i++) {              charIdx[i] = -1;          }            for(int i = 0; i < n; i++) {              if(charIdx[string.charAt(i)] == -1) {                  // character seen first time                  charIdx[string.charAt(i)] = i;              } else {                  // Repeated character                  charIdx[string.charAt(i)] = -2;              }          }            int minIdx = n; // Index of first non repeating character          for(int i = 0; i < 256; i++) {              if(charIdx[i] >= 0 &&                      minIdx > charIdx[i]) {                  minIdx = charIdx[i];              }          }          return (minIdx >= 0 && minIdx < n) ? string.charAt(minIdx) : null;      }        public static Character getFirstNonRepeatingCharacterLinear(String string) {          if(string == null || string.length() == 0) {              return null;          }            int n = string.length();          if(n == 1) {              return string.charAt(0);          }            int[] charCounts = new int[256];          // Initialize character counts of all characters to 0          for(int i = 0; i < 256; i++) {              charCounts[i] = 0;          }            for(int i = 0; i < n; i++) {              charCounts[string.charAt(i)]++;          }            for(int i = 0; i < n; i++) {              if(charCounts[string.charAt(i)] == 1) {                  return string.charAt(i);              }          }          return null;      }        public static Character getFirstNonRepeatingCharacterNaive(String string) {          if(string == null || string.length() == 0) {              return null;          }            int n = string.length();          for(int i = 0; i < n; i++) {              boolean flag = true;              for(int j = 0; j < n; j++) {                  if(i != j && string.charAt(i) == string.charAt(j)) {                      flag = false;                      break;                  }              }              if(flag) {                  return string.charAt(i);              }          }          return null;      }        public static void main(String[] args) {          String string = "ADBCGHIEFKJLADTVDERFSWVGHQWCNOPENSMSJWIERTFB";          System.out.println("Output: " + getFirstNonRepeatingCharacterLinearOptimized(string));      }  } |

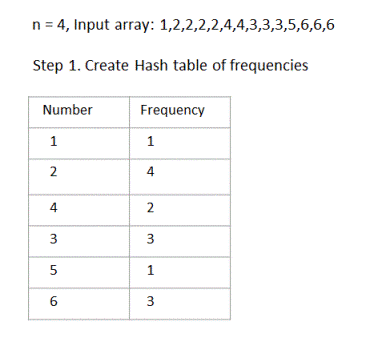
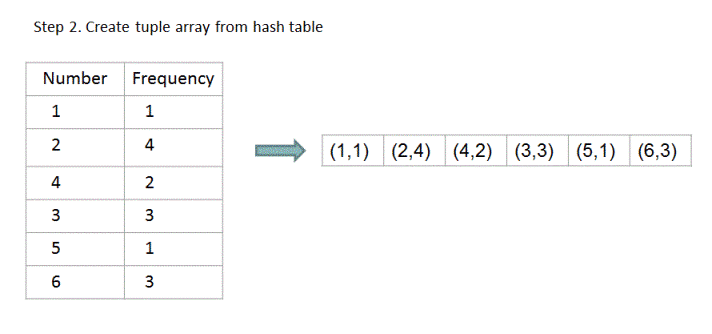
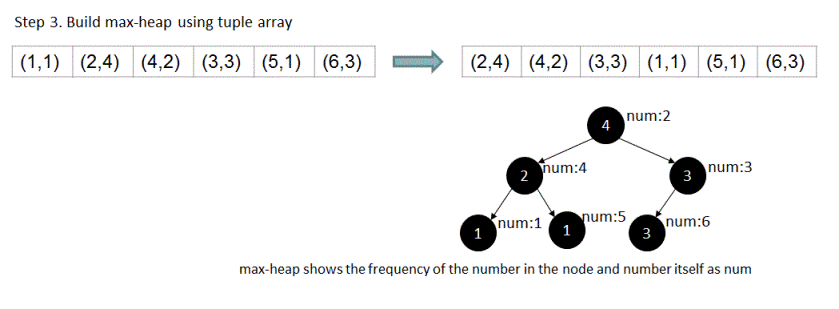
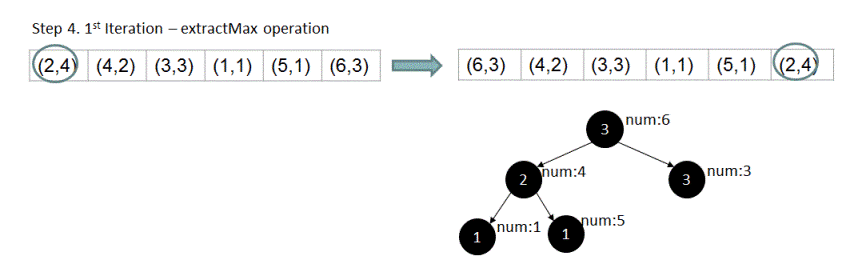
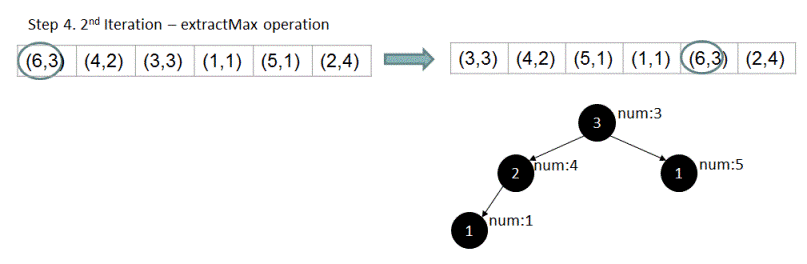
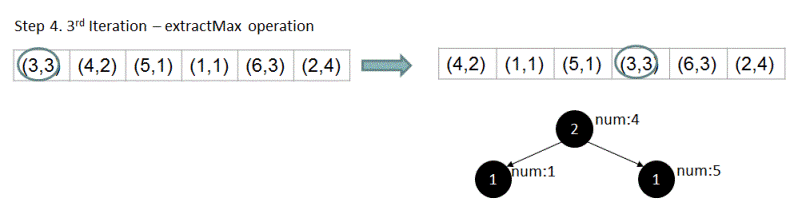
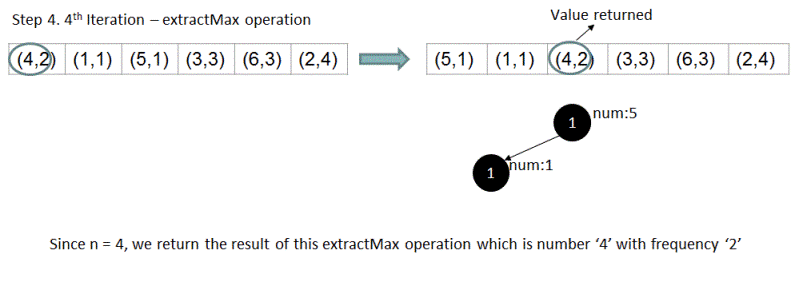
## Order of the Algorithm

Time Complexity is O(n)  
Space Complexity is O(1)

# Find the 'n'th most frequent number in array

Given an array of numbers and a positive integer 'n', find 'n'th most frequent occurring number in that array. If there are more than one numbers which are occurring 'n'th most frequently, then you can return any one of such integers.  
  
Example:  For the input array {1,2,2,2,4,4,4,4,5,5,5,5,5,7,7,8,8,8,8}  
if n = 1, then the output returned should be 5 because it is the most frequent number,  
if n = 2, output can be either 4 or 8 since any of these numbers could be considered as the 2nd most frequent number,  
if n = 3, again output can be either 4 or 8  
if n = 4, output should be 2.

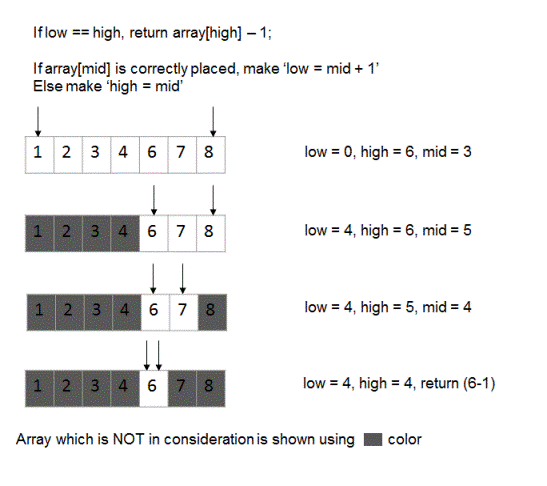
## Algorithm/Insights

**Simple Idea:** One of the simple ideas that could be used is to create a frequency table which stores the frequency of occurrence for each element in the array. Then if we sort this table according to the frequencies, we can easily tell the number which is 'n'th most frequent. If there are 'm' elements in the given array(m > n), frequency table could be created in O(m) time using hash-table and then sorting this table and picking 'n'th most frequent number would take O(mlog(m)) time.  
  
**Optimized Approach:** We can further reduce the time complexity to O(nlog(m)) by making use of max-heap('m' is the length of the input array). This approach would be very similar to heap sort algorithm. For understanding the concept of max-heap and heap sort you can refer [this post.](http://www.ideserve.co.in/learn/heap-sort)  
  
This approach uses following steps -   
1. Using hash table, we first create a frequency table which stores the frequency of occurrence for each number. In this hash table, we define (key, value) tuple as tuple (number 'i', frequency of 'i').  
2. Now we traverse this hash table and create an array which stores these (number, frequency) tuples.  
3. We build the max-heap using this tuple array created in step #2.   
4. By property of max-heap, the root element(or element at 0th index) would be the most frequent element and hence would be the output for input n = 1. For finding out 2nd most frequent element, we swap the root element with the last element and re-arrange the remaining 'm - 1' elements to form a max-heap(let the length of the array be 'm'). Now in this newly created heap(without using the previous root), new root element would again be the largest element in the 'm - 1' elements and hence it would be the 2nd largest element in all 'm' elements.  
5. It follows that if we repeat step #4 'n' times, we should be able to find out 'n'th most frequent number.  
  
If 'm' is the length of the input array then step #1, #2 and #3 take O(m) time. Step #4 takes O(log(m)) time which is executed 'n' number of times hence overall time complexity of this algorithm is O(nlog(m)). Space complexity of this algorithm is O(m).  
  
For array {1,2,2,2,2,4,4,3,3,3,5,6,6,6}, if n = 4, output should be 4 . Let's see how exactly the above algorithm works for this input.  
  
  
  
  
  
  


# Find the missing number in the increasing sequence

Given an increasing sequence of numbers from 1 to n with only one missing number, how can you find that missing number without traversing the sequence in linear fashion. In other words, time complexity of your algorithm must be less than O(n)?  
  
For example, if the given sequence is 1,2,4,5,6,7,8 then the missing number is 3. If the given sequence is 1,3,4,5 then the missing number is 2. For the input 2,3,4,5 output returned should be 1 as it is the missing number.

## Algorithm/Insights

If we traverse the given sequence number by number starting from the element at the 0th index, we can easily tell which number is missing by looking at the difference between the number at index 'i' and number at index 'i-1'. If this difference is greater than 1, we know that we have identified a missing number. For example, for sequence {1,2,3,4,6,7,8}, difference between number 6 and number 4 is 2 and hence we say that missing number is 5. BUT, as per the problem statement we are not supposed to traverse the sequence in linear fashion. Let's look at an optimized approach which runs in log(n) time.  
  
**Correctly placed numbers**: For a number 'i', if all the 'i-1' numbers appearing before it are present in the given sequence then we say that the number 'i' is correctly placed in the given sequence. Because the given sequence is stored in an array which starts from index 0, we can easily check if a number is correctly placed by checking if its index is equal to the value of that number minus 1. For example, for sequence {1,2,3,4,6,7,8}, if you consider number 4 which is correctly placed, its index is 3 which is 4 minus 1. Same is the case for numbers 1,2 and 3. But for numbers 6,7 and 8 which are not correctly placed(since number 5 which appears before them is missing), their respective indices are not equal to their respective values minus 1.   
  
**Idea**: Now using the above idea of correctly placed numbers, to find out the missing number we find the first number from the left in the given sequence(say number 'j') which is incorrectly placed. Once we find this number 'j', we know that the missing number must be 'j-1' since number 'j' is the first element from the left which is incorrectly placed in the given sequence.    
  
**Algorithm**: We find this above defined number 'j' using binary search algorithm. If 'array' stores the given sequence and if function 'findMissingElement(int[] array, int low, int high)' is used to find the missing number in this 'array' with its first element at index 'low' and its last element at index 'high', then the steps used by this function are as following -   
  
1. If low > high, then this is invalid input case. To indicate that, return -1.  
2. If low == high, then we have found the first element which is incorrectly placed. Return 'array[high] - 1' in this case.  
Step #1 and step #2 are base cases for this binary search. Recursive steps are as following.  
  
3. Calculate mid as (low + high)/2.   
a. If array[mid] is correctly placed, then we need to search for the first incorrectly placed element in the higher half of the array(from 'mid + 1' to high). Note that since array[mid] is correctly placed it cannot be the first incorrectly placed element and therefore we exclude it from the search by making low = mid + 1.  
b. If array[mid] is incorrectly placed, then we need to search for the first incorrectly placed element in the lower half of the array(from low to mid). Note that since array[mid] could also be the first element which is incorrectly placed, we cannot exclude it from the search and hence we make high = mid(and not 'mid - 1').  
c. With the modified array boundary in above steps, we make recursive call findMissingElement(array, low, high) to find the missing number.  
  
For more clarity, you may want to check the following diagramatic illustration of the algorithm for the sequence {1,2,3,4,6,7,8}.  


# Find duplicates in an integer array

Given an array of n elements which contains integers from 0 to n-1 only.  
The numbers can appear any number of times. Find the repeating numbers.  
Example:   
Array:  {2, 4, 1, 2, 6, 1, 6, 3, 0}  
Output: [1, 2, 6]

## Algorithm/Insights

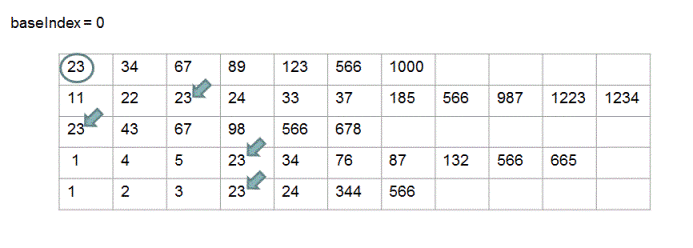
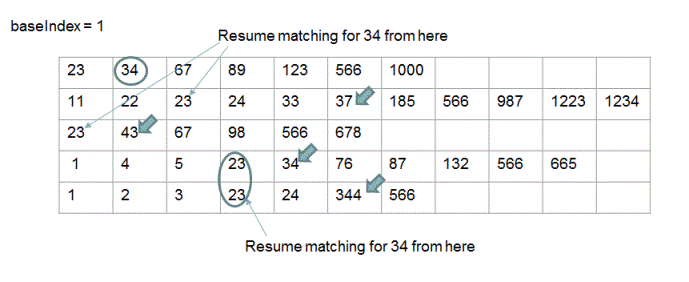
Method 1: Naive Solution  
Use 2 loops to find duplicates in the array. In outer loop, pick each element one by one and in inner loop check if duplicate exists for that element in the array.  
Time Complexity: O(n^2)  
Space Complexity: O(1)  
  
Method 2: Use extra space  
Since the numbers are in the range 1 to n-1, we can create a count array that will store the count of the numbers in the array.  
count[i] represents number of times element i occurs in the array.  
In another traversal, find the elements that have count > 1  
Time Complexity: O(n)  
Space Complexity: O(n)  
  
Method 3:  
Traverse the array once and keep reversing the sign of element at array[i]th position.  
While traversing, if the sign of array[i]th element is already negative, add it to the duplicates list.  
Finally, return duplicates list.  
Time Complexity: O(n)  
Space Complexity: O(1)

|  |
| --- |
| public class FindDuplicates {        public static Set<Integer> getDuplicatesNaive(int[] array) {          Set<Integer> duplicates = new HashSet<Integer>();          for(int i = 0; i < array.length; i++) {              for(int j = i+1; j < array.length; j++) {                  if(array[i] == array[j]) {                      duplicates.add(array[i]);                  }              }          }          return duplicates;      }        public static Set<Integer> getDuplicatesExtraSpace(int[] array) {          int n = array.length;          Set<Integer> duplicates = new HashSet<Integer>();          int[] count = new int[n];            // Initialize count of all numbers to 0          for(int i = 0; i < n; i++) {              count[i] = 0;          }            // Set the count of all elements in count array          for(int i = 0; i < n; i++) {              count[array[i]]++;          }            // Check for duplicates          for(int i = 0; i < n; i++) {              if(count[i] > 1) {                  duplicates.add(i);              }          }          return duplicates;      }        public static Set<Integer> getDuplicates(int[] array) {          int n = array.length;          Set<Integer> duplicates = new HashSet<Integer>();          for(int i = 0; i < n; i++) {              // Make the array element at index array[i] negative if it is positive              if(array[Math.abs(array[i])] > 0) {                  array[Math.abs(array[i])] = -array[Math.abs(array[i])];              } else {                  // If the element at index array[i] is negative, it means we have seen it before, so it is a duplicate                  duplicates.add(Math.abs(array[i]));              }          }          return duplicates;      }        public static void main(String[] args) {          int[] array = {2, 4, 1, 2, 6, 1, 6, 3, 5};          Set<Integer> duplicates = getDuplicates(array);          System.out.println(Arrays.toString(duplicates.toArray()));      }  } |

# Find common elements in 'n' sorted arrays

Given 'n' sorted arrays, find elements which are common in all of these 'n' arrays. For example,   
  
{23, 34, 67, 89, 123, 566, 1000},  
{11, 22, 23, 24,33, 37, 185, 566, 987, 1223, 1234},  
{23, 43, 67, 98, 566, 678},  
{1, 4, 5, 23, 34, 76, 87, 132, 566, 665},  
{1, 2, 3, 23, 24, 344, 566}  
for these given arrays, output should be 23 and 566 since these elements are present in all arrays.                           
  
{1, 3, 4, 4, 5, 43, 67, 98, 566, 678},  
{1, 4, 4, 5, 23, 34, 76, 87, 132, 566, 665},  
{1, 2, 4, 4, 5, 23, 24, 344, 566}  
For above arrays, output should be 1,4,4,5,566,

## Algorithm/Insights

Let's first look at an example to try and understand the algorithm.  
  
In the above shown arrays, we start matching for first element 23 from array-0 in all other arrays. We start looking at elements from index-0 and in this case we find an element in all the other arrays which is equal to first element from array-0. Since this element 23 is present in all arrays, we print it out.  
  
Now the interesting part:  
  
As can be seen above, we now start matching for second element 34 from array-0 in all other arrays. Let's specifically look at array-2 matching process. Instead of starting to match element 34 from index 0, now we can directly start matching for element 34 from the index where we last left the matching process(at index = 2) for this array-2. This is because we know that all the arrays are sorted and therefore all elements to the left of element 23 are less than 23 and we are now trying to match for an element which surely is greater than or equal to 23(since array-0 is also sorted). Same is the case for matching process in array-3, array-4 and so on.    
  
Hopefully, this example has made the algorithm clear.  
  
The algorithm is as following -   
We are given 'n' sorted arrays, say array-1, array-2 to array-n. To find out common elements in these arrays, we pick each element starting from element at index 0 from array-1 and check if that element is present in all remaining arrays. To check if an element from array-0(elementArray0) is present in array-'i'(1 <= i <= n), we start looking at the elements in array-'i' starting from 'last checked index' until we find an element which is greater than or equal to element from array-0(elementArray0). This last checked index would be initialized to 0 for all arrays: array-1 to array-n. We stop when we find an element which is greater than or equal to 'elementArray0', save this index as 'last checked index' for array-'i' and proceed to array-'i+1' for matching element 'elementArray0'. Since array-'i' is sorted, we don't need to check for match from index 0 again when we are trying to match the next element from array-0. This 'last checked index' would be used to resume the matching operation for the next element from array-0.  
  
In the worst case, every element from each array would be visited once and hence time complexity of this algorithm is O(m1 + m2 + m3 + .. + mn) where 'mi' is the length of 'i'th array and total number of arrays is 'n'.  
  
Please checkout function printCommonElements(int[][] arrays) in code snippet for implementation details. Feel free to add comments below in case you have any

|  |
| --- |
| public class CommonElementsSortedArrays  {      public static void printCommonElements(int[][] arrays)      {          if (arrays.length < 2)          {              System.out.println("Too few arrays");              return;          }            // to store the current index for 0th array          int baseIndex = 0;            // to store the current index for each of the remaining n-1 arrays          int[] indices = new int[arrays.length-1];            // to track in how many arrays a specific element is found          int totalMatchFound;            // variable used to terminate the search early          boolean smallestArrayTraversalComplete = false;          /\*           \*  pick elements one by one from the first array           \*  and check we find a match for them in all other arrays           \*/          while ((baseIndex < arrays[0].length) && (!smallestArrayTraversalComplete))          {              totalMatchFound = 0;              for (int i = 1; i < arrays.length; i++)              {                  // get the index for this array where we last stopped                  int currIndex = indices[i-1];                    // skip all the elements in this array which are less than the element at baseIndex in 0th array                  while ((currIndex < arrays[i].length) && (arrays[i][currIndex] < arrays[0][baseIndex]))                  {                      currIndex += 1;                  }                    // check if common element(which is equal to element at baseIndex in 0th array) has been found in this array                  if (currIndex < arrays[i].length)                  {                      if ((arrays[i][currIndex] == arrays[0][baseIndex]))                      {                          totalMatchFound += 1;                      }                  }                  // indicates that we have looked at all the elements of 'i'th array                  else                  {                      smallestArrayTraversalComplete = true;                  }                    // store this incremented index for this array-                  // which would help to resume from this point for next matching round                  indices[i-1] = currIndex;              }                // check if element arrays[0][baseIndex] is found in all other arrays              if (totalMatchFound == arrays.length-1)              {                  System.out.println(arrays[0][baseIndex]);              }              baseIndex += 1;          }      }          public static void main(String[] args)      {          int[][] arrays = {                             {23, 34, 67, 89, 123, 566, 1000},                             {11, 22, 23, 24,33, 37, 185, 566, 987, 1223, 1234},                             {23, 43, 67, 98, 566, 678},                             {1, 4, 5, 23, 34, 76, 87, 132, 566, 665},                             {1, 2, 3, 23, 24, 344, 566}                           };            printCommonElements(arrays);      }  } |

## Order of the Algorithm

Time Complexity is O(m1 + m2 + m3 + .. + mn)  
Space Complexity is O(n)

## Contribution

* Sincere thanks from IDeserve community to Nilesh More for compiling current post.

# Find a Peak Element in an array

Given an array of size n, find a peak element in the array. Peak element is the element which is greater than or equal to its neighbors.  
For example - In Array {1,4,3,6,7,5}, 4 and 7 are peak elements. We need to return any one peak element.

Linear approach:   
One way to solve this problem is to iterate over the array and find an element that is greater than or equal to its neighbors and return it.  
Time Complexity: O(n)  
Efficient approach:  
1. Initialize start = 0, end = array.length - 1  
2. Repeat following steps till peak element is found:  
   a). Find mid = (start+end)/2  
   b). If array[mid] is peak element, return array[mid]  
   c). Else if array[mid-1] > array[mid], find peak in left half of array  
       set end = mid - 1  
   d). Else find peak in right half of array  
       set start = mid + 1  
Time Complexity: O(log n)

public class PeakElement {

    public static Integer getPeakElement(int[] array) {

        if (array == null || array.length == 0) {

            return null;

        }

        int n = array.length;

        int start = 0;

        int end = n - 1;

        while (start <= end) {

            int mid = (start + end) / 2;

            if ((mid == 0 || array[mid - 1] <= array[mid]) && (mid == n - 1 || array[mid] >= array[mid + 1])) {

                return array[mid]; // array[mid] is peak element

            } else if (mid > 0 && array[mid - 1] > array[mid]) {

                end = mid - 1;

            } else {

                start = mid + 1;

            }

        }

        return null;

    }

    public static void main(String[] args) {

        int[] array = { 15, 20, 25, 35, 45, 50, 60 };

        Integer peak = getPeakElement(array);

        System.out.println(peak != null ? "Peak Element is " + peak : "No peak element!");

    }

# Distribute Chocolates Problem

There are M chocolate packets each packet can have variable number of chocolates. There are N students.   
Distribute chocolate packets to students such that:  
a: Each student gets 1 packet  
b: The difference between number of chocolates in packet with maximum chocolates and packet with minimum chocolates given to the students is minimum.

|  |
| --- |
| import java.util.Arrays;    public class DistributeChocolates {        public static void main(String[] args) {          int[] chocolatePackets = {12, 4, 7, 9, 2, 23, 25, 41 , 30, 40, 28, 42, 30, 44, 48, 43, 50};          distributeChocolates(chocolatePackets, 7);      }        public static void distributeChocolates(int[] chocolatePackets, int n) {            // If there are no chocolates or number of students is 0          if(chocolatePackets == null || chocolatePackets.length == 0 || n == 0) {              return;          }            // Sort the chocolatePackets          Arrays.sort(chocolatePackets);          System.out.println(Arrays.toString(chocolatePackets));            int m = chocolatePackets.length;          if(m < n) {              System.out.println("Number of students is more than number of packets. Cannot distribute!");              return;          }          int minDiff = chocolatePackets[m-1];    // Largest number of chocolates          int first = 0;          int last = 0;          int diff = 0;          for(int i = 0; i + n - 1 < m; i++) {              diff = chocolatePackets[i+n-1] - chocolatePackets[i];              if(diff < minDiff) {                  minDiff = diff;                  first = i;                  last = i+n-1;              }          }          System.out.println(chocolatePackets[first] + " " + chocolatePackets[last]);      }  } |

## Order of the Algorithm

Time Complexity is O(n logn)

# Count frequencies of array elements in range 1 to n

Given an array of length n having integers 1 to n with some elements being repeated. Count frequencies of all elements from 1 to n.  
Example:  
Input Array: {2, 3, 3, 2, 5}  
Output:  
1 0  
2 2  
3 2  
4 0  
5 1

Algorithm 1: O(n^2) time and O(1) space  
Use 2 loops.  
1. Outer loop runs from i = 1 to n.  
2. Inner loop calculates the count of i in the input array.  
3. Print the count of i when inner loop ends.  
  
Algorithm 2: O(n) time and O(n) space  
1. Create a count array of size n with all elements from index i = 0 to n-1 initialized to 0.  
Here, count[i] is count of i+1.  
2. Traverse the array once. For i = 0 to n-1, increment count[input[i]-1] by 1.  
3. Traverse count array and print count array.  
  
Algorithm 3: O(n) time and O(1) space  
1. Reduce all elements by 1 so that the elements are converted in the range 0 to n-1.  
2. Traverse the input array and for i = 0 to n-1, add n to element at index (input[i]%n). After all the elements are completed, element at index i will be increment by n\*k where k is the number of times i occurs in the array.  
3. Finally, print counts of elements. Count of i+1 will be input[i]/n.   
Reason:   
Element at index i will be equal to input[i] + n\*k where k is the number of times i occurs in the array.  
On dividing by n, we get: (input[i] + n\*k)/n = input[i]/n + (n\*k)/n = 0 + k = k = the number of times i occurs in the array  
As initially, all elements were reduced by 1, so this is count of i+1.   
4. Traverse input array once and set input[i] = input[i]%n+1 to get original array.  
  
Example:  
Input Array: [2, 3, 3, 2, 5]  
n: 5  
After step #1, array will be:  
[1, 2, 2, 1, 4]  
Step #2:  
i = 0: Set input[input[0]%5] = input[input[0]%5] + 5  
                             = input[1%5] + 5  
                             = input[1] + 5    
                             = 2 + 5   
                             = 7  
Array: [1, 7, 2, 1, 4]  
  
i = 1: Set input[input[1]%5] = input[input[1]%5] + 5  
                             = input[7%5] + 5  
                             = input[2] + 5  
                             = 2 + 5  
                             = 7  
Array: [1, 7, 7, 1, 4]  
  
i = 2: Set input[input[2]%5] = input[input[2]%5] + 5  
                             = input[7%5] + 5  
                             = input[2] + 5  
                             = 7 + 5  
                             = 12  
Array: [1, 7, 12, 1, 4]  
  
i = 3: Set input[input[3]%5] = input[input[3]%5] + 5  
                             = input[1%5] + 5  
                             = input[1] + 5  
                             = 7 + 5  
                             = 12  
Array: [1, 12, 12, 1, 4]  
  
i = 4: Set input[input[4]%5] = input[input[4]%5] + 5  
                             = input[4%5] + 5  
                             = input[4] + 5  
                             = 4 + 5  
                             = 9  
Array: [1, 12, 12, 1, 9]  
  
After step #2: Input Array = [1, 12, 12, 1, 9]  
  
Step #3: Print all elements:  
(i+1)   input[i]/n  
1        0  
2        2  
3        2  
4        0  
5        1  
  
Step #4: Get back original array by setting input[i] = input[i]%n+1  
i = 0: Set input[0] = input[0]%5+1  
                    = 1%5+1  
                    = 1 + 1  
                    = 2   
i = 1: Set input[1] = input[1]%5+1  
                    = 12%5+1  
                    = 2 + 1  
                    = 3   
i = 2: Set input[2] = input[2]%5+1  
                    = 12%5+1  
                    = 2 + 1  
                    = 3   
i = 3: Set input[3] = input[3]%5+1  
                    = 1%5+1  
                    = 1 + 1  
                    = 2   
i = 4: Set input[4] = input[4]%5+1  
                    = 4%5+1  
                    = 4 + 1  
                    = 5   
Hence original array is obtained as [2, 3, 3, 2, 5]

|  |
| --- |
| public class CountFrequencies {        // O(n^2) time and O(1) space      public static void countFrequenciesNaive(int[] input) {          int n = input.length;          for (int i = 1; i <= n; i++) {              int count = 0;              for (int j = 0; j < n; j++) {                  if (input[j] == i) {                      count++;                  }              }              System.out.println(i + " " + count);          }      }        // O(n) time and O(n) space      public static void countFrequencies(int[] input) {          int n = input.length;          int[] count = new int[n];            for (int i = 0; i < n; i++) {              count[i] = 0;          }            for (int i = 0; i < n; i++) {              count[input[i] - 1]++;          }            for (int i = 0; i < n; i++) {              System.out.println(i + 1 + " " + count[i]);          }      }        // O(n) time and O(1) space      public static void countfrequenciesEfficient(int input[]) {            int n = input.length;          for (int i = 0; i < n; i++) {              input[i]--;          }            for (int i = 0; i < n; i++) {              input[input[i] % n] += n;          }            for (int i = 0; i < n; i++) {              System.out.println((i + 1) + " " + input[i] / n);              // Change the element back to original value              input[i] = input[i] % n + 1;          }      }        public static void main(String[] args) {          int[] input = { 2, 3, 3, 2, 5 };          countfrequenciesEfficient(input);      }  } |

## Order of the Algorithm

Time Complexity is O(n)  
Space Complexity is O(1)

# Find all permutations of a String

Given a string, find all the permutations of the string.  
For example:  
Input String: abc  
Output: {bca, acb, abc, cba, bac, cab}

. We use a hash set to store all permutations of the string.  
2. If string is null or of 0 length, we return a hashset with "" as element   
3. We keep first character of the string and recursively call getAllPermutations on the remaining string.   
4. When the function returns, we add the first character to all positions in all the strings that we got in the hashset.  
Please see algorithm visualization to test the algorithm with sample data.

    public static HashSet<String> getAllPermutations(String str) {

        // Create a hash set to prevent any duplicate entries

        HashSet<String> permutations = new HashSet<String>();

        if(str == null || str.length() == 0) {

            permutations.add("");

            return permutations;

        }

        char first = str.charAt(0);

        String remainingString = str.substring(1);

        HashSet<String> words = getAllPermutations(remainingString);

        for(String word: words) {

            for(int i = 0; i <= word.length(); i++) {

                String prefix = word.substring(0, i);

                String suffix = word.substring(i);

                permutations.add(prefix + first + suffix);

            }

        }

        return permutations;

    }

    public static void main(String[] args) {

        String str = "abc";

        HashSet<String> permutations = getAllPermutations(str);

        System.out.println(permutations.toString());

    }

}

# Find pivot in a sorted rotated array

Given a sorted integer array which is rotated any number of times, find the pivot index i.e. index of the minimum element of the array.

Use modified binary search to find pivot element:  
1. If array[0] <= array[length of array - 1], it means the array is not rotated, so return 0.  
2. Initialize start = 0, end = length of array - 1.  
3. Repeat following steps till start <= end   
   a). Set mid = (start+end)/2.  
   b). If mid+1 is pivot, then break.  
   c). If array[start] <= array[mid], it means from start to mid, all elements are in sorted order.  
       Set start = mid+1, so that we look for pivot in second half of the array.  
   d). Else set end = mid-1, to look for pivot in first half.

# Find an element in a sorted rotated array

## Algorithm/Insights

Linear Search:  
    Given an integer array 'array' and an integer 'num'.  
        1. If array is null or is of 0 length, return -1.  
        2. Start with index = 0.  
        3. Check if array[index] == num, then return index.  
        4. Otherwise index++.  
        5. Repeat steps 3 and 4 till index < array length.  
        6. If not found, return -1.  
    Time Complexity: O(n)  
    Space Complexity: O(1)  
  
Using Binary Search:  
    Step 1: Find index of pivot element (minimum element)  
    Step 2: Apply Binary Search on the subarray based on following conditions:  
        1. If num lies between start element and element at pivot-1 position, then find num in array[start..pivot-1] using binary search  
        2. Else if num lies between pivot and last element, then find num in array[pivot..end] using binary search  
      
    Algorithm for finding index of pivot element:  
    1. If array[0] <= array[length of array - 1], it means the array is not rotated, so return 0.  
    2. Initialize start = 0, end = length of array - 1.  
    3. Repeat following steps till start <= end   
       a). Set mid = (start+end)/2.  
       b). If mid+1 is pivot, then break.  
       c). If array[start] <= array[mid], it means from start to mid, all elements are in sorted order. Set start = mid+1, so that we look for pivot in second half of the array.  
       d). Else set end = mid-1, to look for pivot in first half.  
      
    Binary Search algorithm:  
    Start and end are passed as parameter.  
    1. Repeat following steps till start <= end:    
    2. Set mid = (start + end)/2.  
    3. Check if array[mid] == num, then return mid.  
    4. If num < array[mid], set end = mid-1.  
    5. Else set start = mid+1.  
    6. Return -1.

# Find element in sorted rotated array without finding pivot

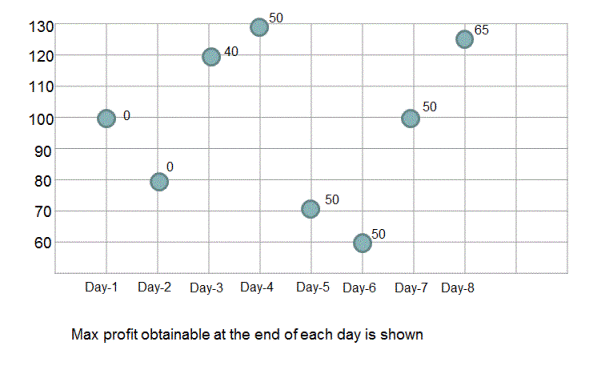
Given a sorted integer array which is rotated any number of times and an integer num, find the index of num in the array. If not found, return -1.

Use modified binary search to find the element  
1. Initialize start = 0, end = array length - 1  
2. Repeat following steps till start <= end:  
3. Set mid = (start + end)/2             
4. If num == array[mid], return mid.  
5. If array[start..mid] is sorted, i.e. array[start] < array[mid]  
    a). If array[start] <= num <= array[mid], set end = mid-1  
    b). Else set start = mid+1  
6. Else If array[mid..end] is sorted, i.e. array[mid] < array[end]  
    a). If array[mid] <= num <= array[end], set start = mid+1  
    b). Else set end = mid-1  
7. If not found, return -1.

# Buy and sell stocks | Part 1

Given an array representing prices of the stock on different days, find the maximum profit that can be earned by performing maximum of one transaction. A transaction consists of activity of buying and selling the stock on different or same days.   
For example in this array - {100, 80, 120, 130, 70, 60, 100, 125} the price of the stock on day-1 is 100, on day-2 is 80 and so on. The maximum profit that could be earned in this window is 65 (buy at 60 and sell at 125).  
  
For price array - {100, 80, 70, 65, 60, 55, 50}, maximum profit that could be earned is 0.

## Algorithm/Insights

For the stock price array - {100, 80, 120, 130, 70, 60, 100, 125}, maximum profit obtainable at the end of each day is shown below.  
  
If you observe above chart carefully, you should be able to notice that: for any given day, maximum profit obtainable is maximum of 1.Maximum profit obtained till previous day,  2.Stock price on that day - minimum stock price so far.  
  
Therefore, to find out the maximum profit obtainable for a given window, all we need to is to keep track of minimum stock price seen so far (starting from day-1) and maximum profit obtained so far. Maximum profit obtained so far can be computed using above observation. Maximum profit obtained so far is initialized to 0 and minimum stock price seen so far is initialized to MAX\_VALUE.   
  
Please checkout function maximumProfit(int[] stockPrices) in code snippet for implementation details.

public class BuyAndSellStocks

{

    public static int maximumProfit(int[] stockPrices)

    {

        int profit = 0;

        int minimumPrice = Integer.MAX\_VALUE;

        /\*

         \* for any given day, maximum profit obtainable is -

         \* maximum of(maximum profit obtained till previous day, stock price on that day - minimum stock price so far)

         \*/

        for(int i = 0; i < stockPrices.length; i++)

        {

            profit = Math.max(profit, stockPrices[i] - minimumPrice);

            minimumPrice = Math.min(stockPrices[i], minimumPrice);

        }

        return profit;

    }

    public static void main(String args[])

    {

        int[] stockPrices = {100, 80, 120, 130, 70, 60, 100, 125};

        System.out.println("maximum profit that could be obtained is: " + maximumProfit(stockPrices));

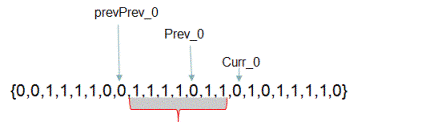
    }

}

# Find index of 0 to replace to get longest continuous sequence of 1s

Given a binary array, find the index of 0 such that when that 0 is replaced with 1 results in longest continuous sequence of 1's. For example, for array {0,1,1,1,0,1,0} replacing 0 at 0th index with 1 results in a sequence of 1's with length 4 and replacing 0 at index 4 with 1 results in a sequence of 1's with length 5. Hence for this input array, output returned should be 4.  
  
For array {0,0,1,1,1,1,0,0,1,1,1,1,0,1,1,0,1,0,1,1,1,1,0} longest sequence of 1's is obtained when we replace 0 at index 12 with 1.

## Algorithm/Insights

The simple idea to solve this problem is to find out number of 1's on the both side of each 0. The 0 which has highest number of 1's around it is the 0 we are looking for. For example for array {0,0,1,1,1,1,0,0,1,1,1,1,0,1,1,0,1,0,1,1,1,1,0}  
number of 1's around 0 with index 0  - 0  
number of 1's around 0 with index 1  - 4  
number of 1's around 0 with index 6  - 4  
number of 1's around 0 with index 7  - 4  
number of 1's around 0 with index 12 - 6  
number of 1's around 0 with index 15 - 3  
number of 1's around 0 with index 17 - 5  
number of 1's around 0 with index 22 - 4  
  
As you can see, if we replace 0 at index 12 with 1 we get sequence of 1's with length as 7 which would be the longest sequence we could get using this array.  
  
Now to get number of 1's around 0, we can simply count them once we visit a 0 while traversing the complete array. This approach would take O(n^2) time.  
  
Another approach to find 1's around each index of 0 uses following technique.   
  
  
If the current index points to 0(say curr\_0) and if we know the index which points to previous occurrence of 0(say prev\_0) and if we also know the index which points to previous to previous occurrence of 0(say prevPrev\_0) then subtracting (prevPrev\_0 + 1) from curr\_0 gives us the number of 1's that could be obtained by replacing 0 with 1 at index prev\_0. This is because between indices prevPrev\_0 and curr\_0 there are all 1's except a single 0 at prev\_0. Please observe above diagram carefully to get complete clarity.  
  
In the function getRequiredIndex() in code snippet, we use same idea to find out the required index of 0. We traverse the complete array and if the current index 'i' points to 0 then we find out the length of longest sequence of 1's that could be obtained by replacing 0 at prev\_0 by calculating: 'i' - (prevPrev\_0 + 1). If this length is greater than maximum such length seen so far then we update maximum length seen so far to new calculated length.  
  
This algorithm runs in O(n) time and with o(1) extra space.

# O(n) time approach to find index of 0 to replace to get longest continuous sequence of 1s

Given a binary array, find the index of 0 such that when that 0 is replaced with 1 results in longest continuous sequence of 1's. For example, for array {0,1,1,1,0,1,0} replacing 0 at 0th index with 1 results in a sequence of 1's with length 4 and replacing 0 at index 4 with 1 results in a sequence of 1's with length 5. Hence for this input array, output returned should be 4.  
  
For array {0,0,1,1,1,1,0,0,1,1,1,1,0,1,1,0,1,0,1,1,1,1,0} longest sequence of 1's is obtained when we replace 0 at index 12 with 1.

## Algorithm/Insights

In the [previous post](http://www.ideserve.co.in/learn/index-of-0-replacing-with-1-results-in-longest-continuous-1s-sequence) we have seen how this can be done in O(n) time and without using extra space. Though that approach is optimized one, we think it's not that intuitive. In this article, we will discuss another more intuitive approach which runs in O(n) time but also requires O(n) extra space.  
  
The basic idea to solve this problem is to find out number of 1's on the both side of each 0. The 0 which has the highest number of 1's around it is the 0 we are looking for. For example for array {0,0,1,1,1,1,0,0,1,1,1,1,0,1,1,0,1,0,1,1,1,1,0}  
number of 1's around 0 with index 0  - 0  
number of 1's around 0 with index 1  - 4  
number of 1's around 0 with index 6  - 4  
number of 1's around 0 with index 7  - 4  
number of 1's around 0 with index 12 - 6  
number of 1's around 0 with index 15 - 3  
number of 1's around 0 with index 17 - 5  
number of 1's around 0 with index 22 - 4  
  
As you can see, if we replace 0 at index 12 with 1 we get sequence of 1's with length as 7 which would be the longest sequence we could get using this array.  
  
In this algorithm, what we do is that we build a count array which would keep track of number of 1's on both side of each 0. For example, countArray[0] = 0 for above example and countArray[1] = 4. Now to build this array, we have to count number of 1's on both side of each 0. To do this, we traverse the input array two times. First time to count number of 1's on left side of each 0 and second time to count number of 1's on right side of each 0. First for loop runs from index 0 to last index and second for loops runs in the opposite direction. Both for loops basically count the number of 1's visited between last seen 0 and current 0 by keeping a running count of number of 1's. At each occurrence of 0, this running count is added in the corresponding countArray, then running count is reset to 0. At each occurrence of 1 this running count is incremented by 1. Both of these for loops run in O(n) time. Once we have the countArray, all we need do is to find out the index of the element with maximum value from this array.

# Find maximum element from each sub-array of size 'k'| Set 1

Given input as an integer array and an integer 'k', find and print element with maximum value from each sub-array of size 'k'.  
For example, for the input array {4,2,12,34,23,35,44,55} and for k = 3, output should be 12,34,34,35,44,55. Observe that 12 is the largest element in the first sub-array {4,2,12}, 34 is the largest element in the second and third sub-arrays - {2,12,34} and {12,34,23} and so on.

## Algorithm/Insights

**Simple approach:** One of the intuitive ways to solve this problem could be to check all the sub-arrays of size 'k' and find out the maximum element from each of these sub-arrays. To check all the sub-arrays of size 'k', we start from a sub-array(window) of the first 'k' elements and then shift this window by one element until we reach the end of the array. There would be total 'n-k+1' such windows of 'k' elements each where 'n' denotes the size of the input array. In the following code snippet, this approach is implemented in function 'simplePrintMaxfromEachSubarray(int[] array, int k)'. Finding maximum element from sub-array of size 'k' takes O(k) time and there are total 'n-k+1' such sub-arrays. Therefore, overall time complexity of this approach is O(k(n-k+1)) which is equivalent to O(nk).  
  
**Optimized approach:** In this approach, we make use of [AVL trees](http://www.ideserve.co.in/learn/avl-tree) (self-balanced Binary Search Tree) as explained in the  following algorithm. We will be referring to AVL tree as a BST in below steps because it would make understanding the intuition of the algorithm easier.  
1. Create a BST from first 'k' elements of the input array.  
2. Find node with maximum value from the BST created in step #1. Print this node's value. This would represent an element with maximum value from first sub-array of size 'k'.  
3. Now starting from i = 0 upto i = n-k-1  
    a. Search and delete element with value array[i] from the BST.  
    b. Insert node with value as array[i+k] into the BST. Now this BST represents next sub-array of size 'k'.  
    c. Find node with maximum value from the BST. Print this node's value.  
  
**Handling duplicates:** Note that this optimized approach which uses BST won't be able to handle the case when input array has more than one elements with same value. We can handle this case easily by pre-processing the input array such that whenever there are duplicates, their values are modified by adding different decimal offsets. For example, if the given input array is {2,3,6,5,6,5} then it could be modified to {2,3,6.01,5.01,6.02,5.02}. Now this modified input does not have any duplicates and above algorithm should work. Care has to taken to remove the decimal part while printing the value of maximum element in a sub-array.  
  
In the above algorithm, creating BST takes O(logk) time. Search and delete operation, insert operation and finding maximum valued node operation, each of these operations takes O(logk) time and we need to perform each of these operations 'n-k' times. Therefore, overall time complexity of this algorithm is O(nlogk). Extra space taken in the form of BST is O(k).  
You can check out function 'printMaxfromEachSubarray(int[] array, int k)' in the following code snippet for implementation details.

# Find maximum element from each sub-array of size 'k'| Set 2

If you are given an integer array and an integer 'k' as input, write a program to print elements with maximum values from each possible sub-array (of given input array) of size 'k'. If the given input array is {9,6,11,8,10,5,14,13,93,14} and for k = 4, output should be 11,11,11,14,14,93,93. Please observe that 11 is the largest element in the first, second and third sub-arrays - {9,6,11,8}, {6,11,8,10} and {11,8,10,5}; 14 is the largest element for fourth and fifth sub-array and 93 is the largest element for remaining sub-arrays.

# Minimum number of coins to make change

Given an infinite supply of coins of values: {C1, C2, ..., Cn} and a sum. Find minimum number of coins that can represent the sum.

## Algorithm/Insights

Consider values set as {2, 5, 3}, n = length of values = 3  
and sum = 7  
Then we can make change for 7 by reducing the given coin values one by one and finding if we can make of the remaining amount.  
For example, for 7, we subtract 2, 5, 3 one by one and then find out if we can make change of the remaining amounts:  
1-> (7-2) = 5,  
2-> (7-5) = 2,  
3-> (7-3) = 4 respectively.  
Of all the possibilities we find the one which gives us minimum number of coins i.e. minimum number of coins from #1, #2 and #3 above and add 1 to it.  
So, the next step in this problem is to find out minimum number of coins to make change for 5, 2, 4 which can be found by applying same strategy as above taking sum as 5, 2 and 4 and finally stopping when no further amount can be reduced from sum or sum becomes 0.  
If no further values can be reduced from sum to make change, and it is still non zero, then this path does not lead to a solution.  
  
Hence, we have following recurrence relation:  
    If sum = 0, minCoins = 0 - because no coins are required to make change for 0.  
    else minCoins(sum) = min(minCoins(sum-values[i])) + 1, for all i = 0 to n-1, where sum >= values[i]  
    If there is no i for which sum >= value[i], then minCoins(sum) = ∞ which is represented by Integer.MAX\_VALUE  
  
For finding minimum number of coins for sum = 7:  
A) minCoins(7)   
               = min(minCoins(7-values[i])) + 1, for all i = 0 to n-1, where sum >= values[i]  
               = min(minCoins(7-2), minCoins(7-5), minCoins(7-3)) + 1  
               = min(minCoins(5), minCoins(2), minCoins(4)) + 1  
  
B) minCoins(5)   
               = min(minCoins(5-values[i])) + 1, for all i = 0 to n-1, where sum >= values[i]  
               = min(minCoins(5-2), minCoins(5-5), minCoins(5-3)) + 1  
               = min(minCoins(3), minCoins(0), minCoins(2)) + 1  
  
C) minCoins(3)   
               = min(minCoins(3-values[i])) + 1, for all i = 0 to n-1, where sum >= values[i]  
               = min(minCoins(3-2), minCoins(3-3)) + 1,             because 3 < 5, we do not consider it  
               = min(minCoins(1), minCoins(0)) + 1  
  
D) minCoins(1) = ∞    because there is no value in set {2, 5, 3} which is less than 1 so 1 cannot be represented using any of these coins.  
  
E) minCoins(0) = 0  
  
Therefore,   
C) minCoins(3)   
               = min(minCoins(1), minCoins(0)) + 1 = min(0, ∞) + 1 = 1  
  
F) minCoins(2)   
               = min(minCoins(2-values[i])) + 1, for all i = 0 to n-1, where sum >= values[i]  
               = min(minCoins(2-2)) + 1  
               = min(minCoins(0)) + 1  
               = 1  
  
Therefore,   
B) minCoins(5)   
               = min(minCoins(3), minCoins(0), minCoins(2)) + 1 = min(1, 0, 1) + 1 = 0 + 1 = 1  
  
G) minCoins(4)   
               = min(minCoins(4-values[i])) + 1, for all i = 0 to n-1, where sum >= values[i]  
               = min(minCoins(4-2), minCoins(4-3)) + 1  
               = min(minCoins(2), minCoins(1)) + 1  
               = min(1, ∞) + 1  
               = 1 + 1  
               = 2  
  
Finally,   
A) minCoins(7)   
               = min(minCoins(5), minCoins(2), minCoins(4)) + 1  
               = min(1, 1, 2) + 1  
               = 2  
  
Time complexity of recursive algorithm is exponential.  
An implementation of this algorithm is provided in the code snippet section.  
  
To solve this problem in a more efficient way, we can use Dynamic Programming.  
As, we can see from above example, that we are recalculating the solution for sub problems again and again. For example, we calculate minCoins(2) again in the sub problems minCoins(4) and minCoins(5).  
If we create an array for already calculated values, then it will save us a lot of recomputation.  
So, we create a minCoins array - minCoins[sum+1] where minCoins[i] represents minimum number of coins required to make change for amount = i.  
We build up the array in bottom up manner starting with minCoins[0].  
So, for any j, if we want to find minCoins[j] then minCoins[0...j-1] will already have been calculated and we just need to find out minimum of minCoins[j-values[i]] where i = 0...n-1 and j >= values[i]  
  
The time complexity of the Dynamic Programming solution is O(n\*sum).  
The space complexity is O(sum).  
Here n is length of the array of values.

public class MinCoins {

    // Recursive Solution to find minimum number of coins

    public static int getMinCoins(int[] values, int sum) {

        if(sum == 0)

            return 0;

        int min = Integer.MAX\_VALUE;

        for(int i = 0; i < values.length; i++) {

            if(sum >= values[i])

                min = Math.min(min, getMinCoins(values, sum-values[i]));

        }

        return min + 1;

    }

    // DP Solution to find minimum number of coins

    public static int getMinCoinsDP(int[] values, int sum) {

        int min = 0;

        int[] minCoins = new int[sum+1];

        minCoins[0] = 0;

        for(int i = 1; i <= sum; i++) {

            min = Integer.MAX\_VALUE;

            for(int j = 0; j < values.length; j++) {

                if(i >= values[j])

                    min = Math.min(min, minCoins[i - values[j]]);

            }

            if(min != Integer.MAX\_VALUE)

                minCoins[i] = min + 1;

            else

                minCoins[i] = Integer.MAX\_VALUE;

        }

        return minCoins[sum];

    }

    public static void main(String[] args) {

        int[] values = {2, 5, 3};

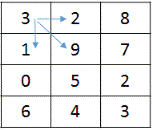
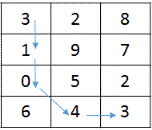
        int sum = 7;

        System.out.println("Minimum number of coins: " + getMinCoinsDP(values, sum));

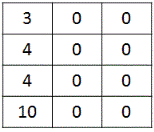
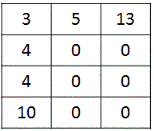
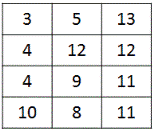
    }

}

# Find minimum cost path in a matrix

Given a cost matrix having a cost at each cell. Find the minimum cost it will take to reach cell (m, n) from top left corner cell (0, 0) if the only allowed directions to move from a cell are right, down and diagonally down.  
  
In this matrix, the minimum cost path to reach cell 3,2 is as shown:  
  
Hence, minimum cost is = 11

## Algorithm/Insights

Consider costMatrix as the given matrix and a position m,n  
Minimum cost to reach m,n from 0,0 will be equal to the cost of cell m,n + total cost to reach cell m,n.  
To reach cell m,n only permissible cells are the cells on left, up and diagonally up. (Because starting from 0,0 we can move further only to right, down and diagonally down cells. So, **cells on left, up and diagonally up are the only cells from which we can reach to a cell.**)  
  
**Recursive solution:**  
1. If m == 0 && n == 0, the costMatrix[0][0] is the solution.  
2. Recursively find minimum cost for cells on left, up and diagonally up direction and add the minimum of these three to current cell cost.  
  
Hence we have these conditions:  
minimumCostPath(costMatrix, m, n) = costMatrix[m][n],      if m == 0 && n == 0  
minimumCostPath(costMatrix, m, n) = costMatrix[m][n] +   
                                                       minimum(minimumCostPath(costMatrix, m - 1, n - 1),  
                                                       minimumCostPath(costMatrix, m - 1, n),  
                                                       minimumCostPath(costMatrix, m, n - 1));  
Time complexity of the recursive algorithm is exponential.  
  
  
**Dynamic Programming solution:**  
The approach followed in recursive solution can be used to efficiently solve this problem by applying dynamic programming.  
Create a minimumCostPath table of size m,n and define:  
minimumCostPath[i][j] = minimum cost to reach (i, j) from (0, 0)  
  
Clearly,   
minimumCostPath[0][0] = costMatrix[0][0]  
minimumCostPath[i][0] = minimumCostPath[i - 1][0] + costMatrix[i][0], for all values of i > 0  
  
minimumCostPath[0][j] = minimumCostPath[0][j - 1] + costMatrix[0][j], for all values of j > 0  
  
  
Next, we fill the minimumCostPath matrix by applying the same formula we used in recursive solution. Since, all the previous values will already have been calculated in the minimumCostPath matrix, we will not have to recalculate these as we did in the recursive solution.  
minimumCostPath[i][j] = costMatrix[i][j] +   
                                           minimum(minimumCostPath[i - 1][j - 1],  
                                           minimumCostPath[i - 1][j],  
                                           minimumCostPath[i][j - 1])  
Here, for calculating minimumCostPath[i][j] we use minimumCostPath[i - 1][j - 1], minimumCostPath[i - 1][j] and minimumCostPath[i][j - 1] because these are the only permissible cells from which we can reach minimumCostPath[i][j].  
  
Finally, we return minimumCostPath[m][n].

|  |
| --- |
| public class MinimumCostPath {        public static int minimumCostPathRec(int[][] costMatrix, int m, int n) {          if (m < 0 || n < 0)              return Integer.MAX\_VALUE;            if (m == 0 && n == 0)              return costMatrix[0][0];            return costMatrix[m][n]                  + minimum(minimumCostPathRec(costMatrix, m - 1, n - 1),                            minimumCostPathRec(costMatrix, m - 1, n),                            minimumCostPathRec(costMatrix, m, n - 1));      }        public static int minimumCostPathDP(int[][] costMatrix, int m, int n) {          int[][] minimumCostPath = new int[m+1][n+1];          minimumCostPath[0][0] = costMatrix[0][0];            for (int i = 1; i <= m; i++) {              minimumCostPath[i][0] = minimumCostPath[i - 1][0] + costMatrix[i][0];          }            for (int j = 1; j <= n; j++) {              minimumCostPath[0][j] = minimumCostPath[0][j - 1] + costMatrix[0][j];          }            for (int i = 1; i <= m; i++) {              for (int j = 1; j <= n; j++) {                  minimumCostPath[i][j] = costMatrix[i][j]                                          + minimum(minimumCostPath[i - 1][j - 1],                                                    minimumCostPath[i - 1][j],                                                    minimumCostPath[i][j - 1]);              }          }          return minimumCostPath[m][n];      }        public static int minimum(int a, int b, int c) {          return Math.min(a, Math.min(b, c));      }        public static void main(String[] args) {          int[][] costMatrix = { { 3, 2, 8 }, { 1, 9, 7 }, { 0, 5, 2 }, {6, 4, 3} };          System.out.println("Minimum cost: " + minimumCostPathDP(costMatrix, 3, 2));      }  } |

## Order of the Algorithm

Time Complexity is O(mn)  
Space Complexity is O(mn)

# Find maximum value of sum of index element products(i\*array[i]) with only rotations allowed on a given array

Given an array, find the maximum possible value of sum of index-element-products(i\*array[i]) with only rotations allowed on a given array. Sum of index-element-products for array of length 'n' is computed as - 0\*array[0] + 1\*array[1] + 2\*array[2] + ... + n-1\*array[n-1].  
  
For example, for the array {3,4,5,6,1,2} without doing any rotations sum of index-element-products is 46. After doing one clockwise rotation of the array, it would be modified to {2,3,4,5,6,1} and sum of index-element-products in this case is 55.   
As you should be able to confirm, maximum value of sum of index-element-products for this given array is 70 which is obtained after performing two clockwise rotations in which case modified array is {1,2,3,4,5,6}.  
  
For the array {24,26,25,22},   
index-element-products sum without any rotation is 142. The maximum sum of index-element-products that could be obtained is 151 which is obtained with one clockwise rotation(modified array {22,24,26,25}).

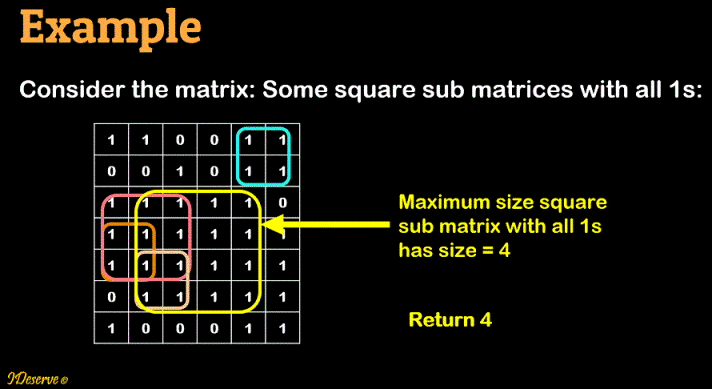
## Video coming soon!

#### Subscribe for more updates

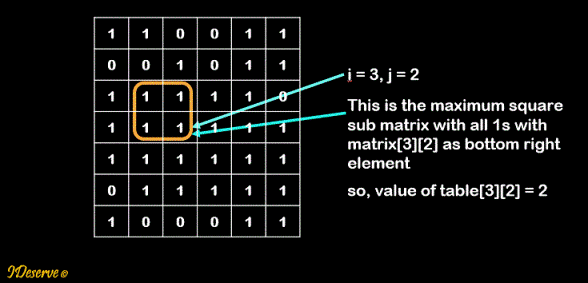
## Algorithm/Insights

**Simple approach:** If 'n' is the length of given array and if we perform all the 'n' possible clockwise rotations then we can compute all possible values of sum of index-element-products to find out maximum possible sum. Since computing sum of index-element-products for an array takes O(n) time, overall time complexity of this approach is O(n^2). You can check function 'simpleFindMaxIndexElementProductSum(int[] array)' in following code snippet for implementation details..  
  
**Optimized approach:** Without doing any rotations on the given 'array', sum of index-element-products is -   
0\*array[0] + 1\*array[1] + 2\*array[2] + .... + (n-2)\*array[n-2] + (n-1)\*array[n-1]. Let's call this sum as 'sum\_0'   
  
After performing one clockwise rotation, sum of index-element-products would be -  
0\*array[n-1] + 1\*array[0] + 2\*array[1] + .... + (n-2)\*array[n-3] + (n-1)\*array[n-2]. Let's call this sum as 'sum\_1'.  
  
After performing two clockwise rotations, sum of index-element-products would be -  
0\*array[n-2] + 1\*array[n-1] + 2\*array[0] + .... + (n-2)\*array[n-4] + (n-1)\*array[n-3]. Let's call this sum as 'sum\_2'.  
  
Now the value of 'sum\_1' - 'sum\_0' is (array[0] + array[1] + .. array[n-2]) - (n-1)\*array[n-1] which is equal to (array[0] + array[1] + .. array[n-2] + array[n-1]) - n\*array[n-1].  
Similarly, value of 'sum\_2' - 'sum\_1' is (array[0] + array[1] + .. array[n-2] + array[n-1]) - n\*array[n-2]  
  
Using the same analogy above, given 'sum\_(i-1)'(sum after 'i-1' clockwise rotations) we can compute 'sum\_i' by using   
sum\_i = sum\_(i-1) + sum of all array elements - n\*(array[n-i]).  
  
If we use this mathematical approach, we need to iterate over all the array elements only once to compute 'sum\_0' and sum of all array elements. Using these values and above mathematical relation, we can then compute values of sum of index-element-products for all the 'n' possible clockwise rotations of given array and find out the maximum sum possible in O(n) time. Therefore, overall time complexity of this approach is O(n) with O(1) space complexity. In the code snippet below, you can check the function 'findMaxIndexElementProductSum(int[] array)' for implementation details.

# Maximum size square sub-matrix with all 1s

Given a matrix of dimensions mxn having all entries as 1 or 0, find out the size of maximum size square sub-matrix with all 1s.  


## Algorithm/Insights

Step 1:  Create an empty table of size mxn, defined as  
    table[i][j] = Size of maximum square sub matrix with all 1s with matrix[i][j] as bottom right element.  
      
  
Step 2: Set values in table based on the conditions  
    a. if i = 0 or j = 0, table[i][j] = matrix[i][j]  
    b. else if matrix[i][j] = 0 then table[i][j] = 0  
    c. else table[i][j] = min(table[i - 1][j - 1], table[i - 1][j], table[i][j - 1]) + 1;  
  
During step 2, maintain largest element added in the table and finally return it.

|  |
| --- |
| public class MaximumSizeSquareSubmatrixWithAllOnes {        private static int maximumSizeSquareSubmatrixWithAllOnes(int[][] matrix) {          int maxSize = 0;          int r = matrix.length;          int c = matrix[0].length;          // Step 1          int[][] table = new int[r][c];            for (int i = 0; i < r; i++) {              for (int j = 0; j < c; j++) {                  // Step 2 a                  if (i == 0 || j == 0) {                      table[i][j] = matrix[i][j];                      maxSize = table[i][j] > maxSize ? table[i][j] : maxSize;                  }                  // Step 2 b                  else if (matrix[i][j] == 0) {                      table[i][j] = 0;                  }                  // Step 2 c                  else {                      table[i][j] = min(table[i - 1][j - 1], table[i - 1][j], table[i][j - 1]) + 1;                      maxSize = table[i][j] > maxSize ? table[i][j] : maxSize;                  }              }          }            return maxSize;      }        private static int min(int i, int j, int k) {          return i <= j && i <= k ? i : (j <= i && j <= k ? j : k);      }        public static void main(String[] args) {          int matrix[][] = { { 1, 1, 0, 0, 1, 1 },                             { 0, 0, 1, 0, 1, 1 },                             { 1, 1, 1, 1, 1, 0 },                             { 1, 1, 1, 1, 1, 1 },                             { 1, 1, 1, 1, 1, 1 },                             { 0, 1, 1, 1, 1, 1 },                             { 1, 0, 0, 0, 1, 1 }                           };          System.out.println(maximumSizeSquareSubmatrixWithAllOnes(matrix));      }  } |

## Order of the Algorithm

Time Complexity is O(mn)  
Space Complexity is O(mn)

## Contribution

# Longest Subset With Consecutive Numbers

Given a set of numbers, find the longest subset of consecutive numbers.  
  
Example:  
Input:  
1 3 8 14 4 10 2 11  
Output:  
1 2 3 4

# Longest Increasing Subsequence O(n logn)

Given an array of integers, find the longest increasing subsequence.   
Example: X = {3,1,5,2,6,4,9} LIS(X) = {1,2,4,9}

|  |
| --- |
| public class LongestIncreasingSubsequence {        public static void LIS(int X[])      {          int parent[]= new int[X.length]; //Tracking the predecessors/parents of elements of each subsequence.          int increasingSub[]= new int[X.length + 1]; //Tracking ends of each increasing subsequence.          int length = 0; //Length of longest subsequence.            for(int i=0; i<X.length; i++)          {              //Binary search              int low = 1;              int high = length;              while(low <= high)              {                  int mid = (int) Math.ceil((low + high)/2);                    if(X[increasingSub[mid]] < X[i])                      low = mid + 1;                  else                      high = mid - 1;              }                int pos = low;              //update parent/previous element for LIS              parent[i] = increasingSub[pos-1];              //Replace or append              increasingSub[pos] =  i;                //Update the length of the longest subsequence.              if(pos > length)                  length=pos;          }            //Generate LIS by traversing parent array          int LIS[] = new int[length];          int k   = increasingSub[length];          for(int j=length-1; j>=0; j--)          {              LIS[j] =  X[k];              k = parent[k];          }              for(int i=0; i<length; i++)          {              System.out.println(LIS[i]);          }          }        public static void main(String args[])      {          int X[] = {3,1,5,0,6,4,9};          LIS(X);      }  } |

## Order of the Algorithm

Time Complexity is O(nlog n)  
Space Complexity is O(n)

A city's skyline is the outer contour of the silhouette formed by all the buildings in that city when viewed from a distance.  
     
Now suppose you are given the locations and heights of all the buildings, write a program to output the skyline formed by these buildings collectively. Location of building is specified using tuple [x1,x2,h] where 'x1' is starting point of a building, 'x2' is where building ends and 'h' is height of the building. The skyline would be returned as a set of key points.   
Starting point of each horizontal line segment of the skyline is marked as a key point. Complete skyline therefore could be specified using set of such key points.   
  
For example, if input is an array of building co-ordinates: [[2,9,10], [13,15,10]] then the output should be the skyline specified as [[2,10],[9,0],[13,10],[15,0]]. In the below diagram, input buildings are shown on the left hand side and the  output is shown on the right hand side with highlighted key points.

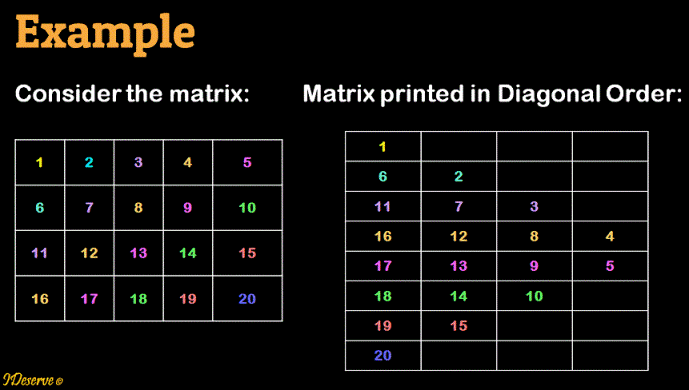
================================================

# Find the length of longest increasing subsequence in an array

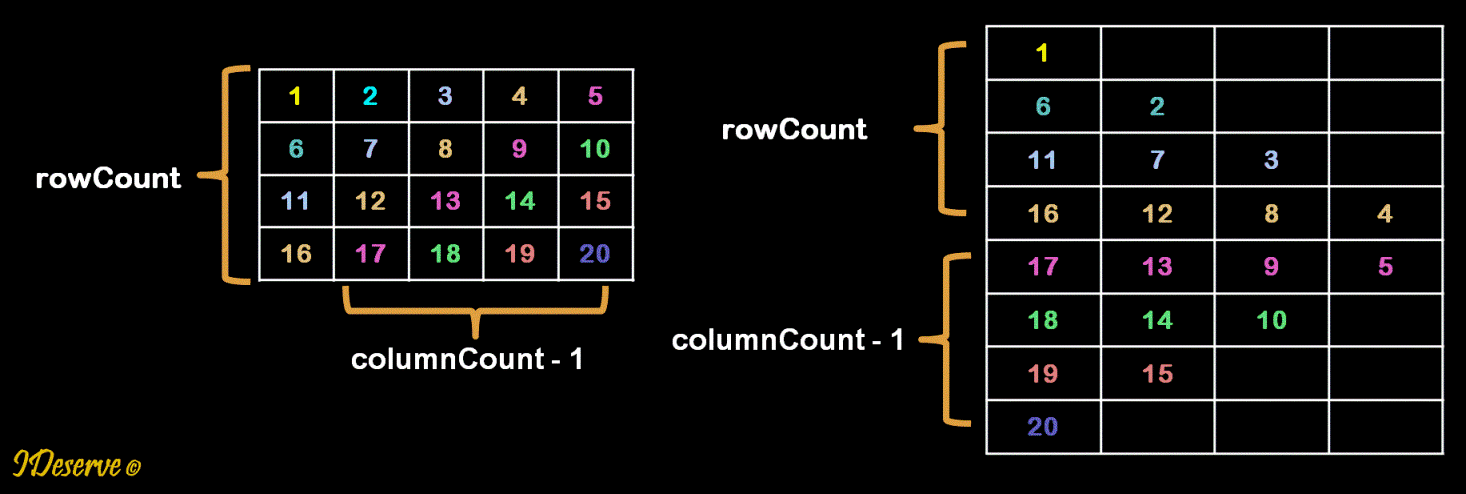
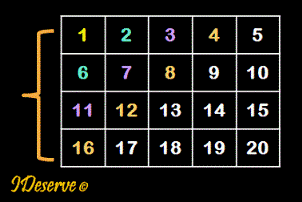
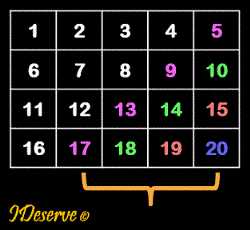
Find the length of longest increasing subsequence in an array.  
Example:  
For input array:  {12, 18, 7, 34, 30, 28, 90, 88}  
length of longest increasing subsequence is 4.

This problem can be solved by applying dynamic programming strategy as explained below:  
1. Create an array lisLength, of length same as input array length, to store longest increasing subsequence defined as:  
   lisLength[i] = length of longest increasing subsequence in the array [0..i] including ith element  
2. Initialize lisLength[i] to 1.  
3. The lisLength array is then updated one element by element:  
   lisLength[i] = Max(lisLength[j]) + 1, for all values of j where input[j] < input[i] and j < i  
  
How this works:  
For j = 0 to i-1, if input[j] < input[i], then input[i] can be added to the longest increasing subsequence which is formed from array elements 0 to j.  
Therefore, to find longest increasing subsequence in 0..i, lisLength[i], we need the Max(lisLength[j]) value which satisfies this condition.  
If there is no such j, then longest increasing subsequence in input array 0..i, lisLength[i] = 1 which is formed by adding only ith element.  
  
Example:  
http://www.ideserve.co.in/learn/img/length-of-longest-bitonic-subsequence-in-an-array_1.gif  
  
lisLength[] : [1, 0, 0, 0, 0, 0, 0, 0]  
  
i = 1: lisLength[1] = 1  
    j = 0: input[0] = 12 < input[1] = 18, also lisLength[1] = 1 < lisLength[0] + 1 = 2, so we update:  
        lisLength[1] = lisLength[0] + 1 = 2  
lisLength[] : [1, 2, 0, 0, 0, 0, 0, 0]  
  
i = 2: lisLength[2] = 1  
    j = 0: input[0] = 12 > input[2] = 7  
    j = 1: input[1] = 18 > input[2] = 7  
lisLength[] : [1, 2, 1, 0, 0, 0, 0, 0]  
  
i = 3: lisLength[3] = 1  
    j = 0: input[0] = 12 < input[3] = 34, also lisLength[3] = 1 < lisLength[0] + 1 = 2, so we update:  
        lisLength[3] = lisLength[0] + 1 = 2  
    j = 1: input[1] = 18 < input[3] = 34, also lisLength[3] = 2 < lisLength[1] + 1 = 3, so we update:  
        lisLength[3] = lisLength[1] + 1 = 3  
    j = 2: input[2] =  7 < input[3] = 34, but lisLength[3] = 3 > lisLength[2] + 1 = 2, so lisLength is not updated.  
lisLength[] : [1, 2, 1, 3, 0, 0, 0, 0]  
  
i = 4: lisLength[4] = 1  
    j = 0: input[0] = 12 < input[4] = 30, also lisLength[4] = 1 < lisLength[0] + 1 = 2, so we update:  
        lisLength[4] = lisLength[0] + 1 = 2  
    j = 1: input[1] = 18 < input[4] = 30, also lisLength[4] = 2 < lisLength[1] + 1 = 3, so we update:  
        lisLength[4] = lisLength[1] + 1 = 3  
    j = 2: input[2] =  7 < input[4] = 30, but lisLength[4] = 3 > lisLength[2] + 1 = 2, so lisLength is not updated.  
    j = 3: input[3] = 34 > input[4] = 30  
lisLength[] : [1, 2, 1, 3, 3, 0, 0, 0]  
  
i = 5: lisLength[5] = 1  
    j = 0: input[0] = 12 < input[5] = 28, also lisLength[5] = 1 < lisLength[0] + 1 = 2, so we update:  
        lisLength[5] = lisLength[0] + 1 = 2  
    j = 1: input[1] = 18 < input[5] = 28, also lisLength[5] = 2 < lisLength[1] + 1 = 3, so we update:  
        lisLength[5] = lisLength[1] + 1 = 3  
    j = 2: input[2] =  7 < input[5] = 28, but lisLength[5] = 3 > lisLength[2] + 1, so lisLength is not updated.  
    j = 3: input[3] = 34 > input[5] = 28  
    j = 4: input[4] = 30 > input[5] = 28  
lisLength[] : [1, 2, 1, 3, 3, 3, 0, 0]  
  
i = 6: lisLength[6] = 1  
    j = 0: input[0] = 12 < input[6] = 90, also lisLength[6] = 1 < lisLength[0] + 1 = 2, so we update:  
        lisLength[6] = lisLength[0] + 1 = 2  
    j = 1: input[1] = 18 < input[6] = 90, also lisLength[6] = 2 < lisLength[1] + 1 = 3, so we update:  
        lisLength[6] = lisLength[1] + 1 = 3  
    j = 2: input[2] =  7 < input[6] = 90, but lisLength[6] = 3 > lisLength[2] + 1, so lisLength is not updated.  
    j = 3: input[3] = 34 < input[6] = 90, also lisLength[6] = 3 < lisLength[3] + 1 = 4, so we update:  
        lisLength[6] = lisLength[3] + 1 = 4  
    j = 4: input[4] = 30 < input[6] = 90, but lisLength[6] = 4 = lisLength[4] + 1, so lisLength is not updated.  
    j = 5: input[5] = 28 < input[6] = 90, but lisLength[6] = 4 = lisLength[5] + 1, so lisLength is not updated.  
lisLength[] : [1, 2, 1, 3, 3, 3, 4, 0]  
  
i = 7: lisLength[7] = 1  
    j = 0: input[0] = 12 < input[7] = 88, also lisLength[7] = 1 < lisLength[0] + 1 = 2, so we update:  
        lisLength[7] = lisLength[0] + 1 = 2  
    j = 1: input[1] = 18 < input[7] = 88, also lisLength[7] = 1 < lisLength[1] + 1 = 3, so we update:  
        lisLength[7] = lisLength[1] + 1 = 3  
    j = 2: input[2] =  7 < input[7] = 88, but lisLength[7] = 3 > lisLength[2] + 1, so lisLength is not updated.  
    j = 3: input[3] = 34 < input[7] = 88, also lisLength[7] = 3 < lisLength[3] + 1 = 4, so we update:  
        lisLength[7] = lisLength[3] + 1 = 4  
    j = 4: input[4] = 30 < input[7] = 88, but lisLength[7] = 4 = lisLength[4] + 1, so lisLength is not updated.  
    j = 5: input[5] = 28 < input[7] = 88, but lisLength[7] = 4 = lisLength[5] + 1, so lisLength is not updated.  
    j = 6: input[6] = 90 > input[7] = 88  
lisLength[] : [1, 2, 1, 3, 3, 3, 4, 4]  
  
Solution: Maximum value in lisLength array = 4.  
  
Time complexity of this solution is O(n^2)

# Print a Matrix Diagonally

Given a matrix of mxn dimensions, print the elements of the matrix in diagonal order.  
  
Please check out the video below for detailed explanation of the algorithm with animations.

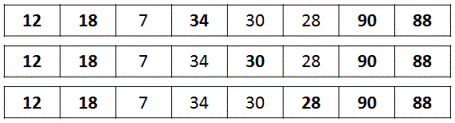
## Algorithm/Insights

rowCount = number of rows  
columnCount = number of columns  
Then, number of diagonals will be = rowCount + columnCount - 1 as depicted in the diagram below.  
  
  
Step 1: Print first rowCount diagonals   
Print diagonals that start from the first column   
elements.  
  
  
Step 2: Print next columnCount - 1 diagonals   
Print diagonals that start from the last row   
elements.  
  
  
Step 1 details: Print first rowCount diagonals   
Iterate to print diagonals from row k = 0 to rowCount - 1.  
1: Start with row = k and col = 0  
2: Print the element matrix[row][col]  
3: Decrement row by 1, increment col by 1 till row >= 0 and  col < columnCount  
  
Step 2 details: Print next columnCount - 1 diagonals   
Iterate to print diagonals from column k = 1 to columnCount - 1  
1: Start with last row, row = rowCount - 1 and col = k  
2: Print the element matrix[row][col]  
3: Decrement row by 1, increment col by 1 till row >= 0 and  col < columnCount

# Find the length of longest bitonic subsequence in an array

Given an array of size n, find the longest bitonic subsequence in the array.  
A bitonic sequence a sequence which is first increasing and then decreasing.  
i.e. it is of the form:  
x1, x2, x3, ... xn  
where:  
    x1 < x2 < x3 < ... < xm  
    xm > xm+1 > xm+2 > ... > xn  
For example: 10, 25, 36, 40, 59, 48, 34, 20, 5  
Here the sequence first increases from 10 to 59 then descreases to 5.

## Algorithm/Insights

The problem can be solved by using longest increasing subsequence strategy.  
Please go through this post on finding longest increasing subsequence:  
[Longest Increasing Subsequence](http://www.ideserve.co.in/learn/longest-increasing-subsequence)  
  
Algorithm for finding longest bitonic subsequence:  
1. Given input array of length n.  
2. Create array lisLength[] where lisLength[i] = Length of Longest Increasing Subsequence in input[0..i] including ith element  
3. Create array ldsLength[] where ldsLength[i] = Length of Longest Decreasing Subsequence in input[i..n-1] including ith element  
Then length of longest increasing bitonic subsequence which includes the ith element can be calculated as:  
        lisLength[i] + ldsLength[i] - 1  
Here 1 is subtracted from the sum of lengths because ith element is counted twice.  
Therefore, length of longest bitonic subsequence is the maximum value for i = 0..n-1  
        max(lisLength[i] + ldsLength[i] - 1)  
  
Example:  
input array:  
http://www.ideserve.co.in/learn/img/length-of-longest-bitonic-subsequence-in-an-array_1.gif  
  
lisLength[] : [1, 0, 0, 0, 0, 0, 0, 0]  
  
i = 1: lisLength[1] = 1  
    j = 0: input[0] = 12 < input[1] = 18, also lisLength[1] = 1 < lisLength[0] + 1 = 2, so we update:  
        lisLength[1] = lisLength[0] + 1 = 2  
lisLength[] : [1, 2, 0, 0, 0, 0, 0, 0]  
  
i = 2: lisLength[2] = 1  
    j = 0: input[0] = 12 > input[2] = 7  
    j = 1: input[1] = 18 > input[2] = 7  
lisLength[] : [1, 2, 1, 0, 0, 0, 0, 0]  
  
i = 3: lisLength[3] = 1  
    j = 0: input[0] = 12 < input[3] = 34, also lisLength[3] = 1 < lisLength[0] + 1 = 2, so we update:  
        lisLength[3] = lisLength[0] + 1 = 2  
    j = 1: input[1] = 18 < input[3] = 34, also lisLength[3] = 2 < lisLength[1] + 1 = 3, so we update:  
        lisLength[3] = lisLength[1] + 1 = 3  
    j = 2: input[2] =  7 < input[3] = 34, but lisLength[3] = 3 > lisLength[2] + 1 = 2, so lisLength is not updated.  
lisLength[] : [1, 2, 1, 3, 0, 0, 0, 0]  
  
i = 4: lisLength[4] = 1  
    j = 0: input[0] = 12 < input[4] = 30, also lisLength[4] = 1 < lisLength[0] + 1 = 2, so we update:  
        lisLength[4] = lisLength[0] + 1 = 2  
    j = 1: input[1] = 18 < input[4] = 30, also lisLength[4] = 2 < lisLength[1] + 1 = 3, so we update:  
        lisLength[4] = lisLength[1] + 1 = 3  
    j = 2: input[2] =  7 < input[4] = 30, but lisLength[4] = 3 > lisLength[2] + 1 = 2, so lisLength is not updated.  
    j = 3: input[3] = 34 > input[4] = 30  
lisLength[] : [1, 2, 1, 3, 3, 0, 0, 0]  
  
i = 5: lisLength[5] = 1  
    j = 0: input[0] = 12 < input[5] = 28, also lisLength[5] = 1 < lisLength[0] + 1 = 2, so we update:  
        lisLength[5] = lisLength[0] + 1 = 2  
    j = 1: input[1] = 18 < input[5] = 28, also lisLength[5] = 2 < lisLength[1] + 1 = 3, so we update:  
        lisLength[5] = lisLength[1] + 1 = 3  
    j = 2: input[2] =  7 < input[5] = 28, but lisLength[5] = 3 > lisLength[2] + 1, so lisLength is not updated.  
    j = 3: input[3] = 34 > input[5] = 28  
    j = 4: input[4] = 30 > input[5] = 28  
lisLength[] : [1, 2, 1, 3, 3, 3, 0, 0]  
  
i = 6: lisLength[6] = 1  
    j = 0: input[0] = 12 < input[6] = 90, also lisLength[6] = 1 < lisLength[0] + 1 = 2, so we update:  
        lisLength[6] = lisLength[0] + 1 = 2  
    j = 1: input[1] = 18 < input[6] = 90, also lisLength[6] = 2 < lisLength[1] + 1 = 3, so we update:  
        lisLength[6] = lisLength[1] + 1 = 3  
    j = 2: input[2] =  7 < input[6] = 90, but lisLength[6] = 3 > lisLength[2] + 1, so lisLength is not updated.  
    j = 3: input[3] = 34 < input[6] = 90, also lisLength[6] = 3 < lisLength[3] + 1 = 4, so we update:  
        lisLength[6] = lisLength[3] + 1 = 4  
    j = 4: input[4] = 30 < input[6] = 90, but lisLength[6] = 4 = lisLength[4] + 1, so lisLength is not updated.  
    j = 5: input[5] = 28 < input[6] = 90, but lisLength[6] = 4 = lisLength[5] + 1, so lisLength is not updated.  
lisLength[] : [1, 2, 1, 3, 3, 3, 4, 0]  
  
i = 7: lisLength[7] = 1  
    j = 0: input[0] = 12 < input[7] = 88, also lisLength[7] = 1 < lisLength[0] + 1 = 2, so we update:  
        lisLength[7] = lisLength[0] + 1 = 2  
    j = 1: input[1] = 18 < input[7] = 88, also lisLength[7] = 1 < lisLength[1] + 1 = 3, so we update:  
        lisLength[7] = lisLength[1] + 1 = 3  
    j = 2: input[2] =  7 < input[7] = 88, but lisLength[7] = 3 > lisLength[2] + 1, so lisLength is not updated.  
    j = 3: input[3] = 34 < input[7] = 88, also lisLength[7] = 3 < lisLength[3] + 1 = 4, so we update:  
        lisLength[7] = lisLength[3] + 1 = 4  
    j = 4: input[4] = 30 < input[7] = 88, but lisLength[7] = 4 = lisLength[4] + 1, so lisLength is not updated.  
    j = 5: input[5] = 28 < input[7] = 88, but lisLength[7] = 4 = lisLength[5] + 1, so lisLength is not updated.  
    j = 6: input[6] = 90 > input[7] = 88  
lisLength[] : [1, 2, 1, 3, 3, 3, 4, 4]  
  
Similarly, longest decreasing subsequence length array is computed as follows:  
  
ldsLength[] : [0, 0, 0, 0, 0, 0, 0, 1]  
  
i = n-2 = 6: ldsLength[6] = 1  
    j = n-1 = 7: input[7] = 88 < input[6] = 90  
       ldsLength[6] = ldsLength[7] + 1 = 2  
ldsLength[] : [0, 0, 0, 0, 0, 0, 2, 1]  
  
i = 5: ldsLength[5] = 1  
    j = 7: input[7] = 88 > input[5] = 28  
    j = 6: input[6] = 90 > input[5] = 28  
ldsLength[] : [0, 0, 0, 0, 0, 1, 2, 1]  
  
i = 4: ldsLength[4] = 1  
    j = 7: input[7] = 88 > input[5] = 30  
    j = 6: input[6] = 90 > input[5] = 30  
    j = 5: input[5] = 28 < input[5] = 30  
       ldsLength[4] = ldsLength[5] + 1 = 2  
ldsLength[] : [0, 0, 0, 0, 2, 1, 2, 1]  
  
i = 3: ldsLength[3] = 1  
    j = 7: input[7] = 88 > input[5] = 34  
    j = 6: input[6] = 90 > input[5] = 34  
    j = 5: input[5] = 28 < input[5] = 34  
       ldsLength[3] = ldsLength[5] + 1 = 2  
    j = 4: input[4] = 30 < input[5] = 34  
       ldsLength[3] = ldsLength[4] + 1 = 3  
ldsLength[] : [0, 0, 0, 3, 2, 1, 2, 1]  
  
i = 2: ldsLength[2] = 1  
    j = 7: input[7] = 88 > input[5] = 7  
    j = 6: input[6] = 90 > input[5] = 7  
    j = 5: input[5] = 28 > input[5] = 7  
    j = 4: input[4] = 30 > input[5] = 7  
    j = 3: input[3] = 34 > input[5] = 7  
ldsLength[] : [0, 0, 1, 3, 2, 1, 2, 1]  
  
i = 1: ldsLength[1] = 1  
    j = 7: input[7] = 88 > input[5] = 18  
    j = 6: input[6] = 90 > input[5] = 18  
    j = 5: input[5] = 28 > input[5] = 18  
    j = 4: input[4] = 30 > input[5] = 18  
    j = 3: input[3] = 34 > input[5] = 18  
    j = 2: input[2] =  7 < input[5] = 18  
       ldsLength[2] = ldsLength[2] + 1 = 2  
ldsLength[] : [0, 2, 1, 3, 2, 1, 2, 1]  
  
i = 0: ldsLength[0] = 1  
    j = 7: input[7] = 88 > input[5] = 12  
    j = 6: input[6] = 90 > input[5] = 12  
    j = 5: input[5] = 28 > input[5] = 12  
    j = 4: input[4] = 30 > input[5] = 12  
    j = 3: input[3] = 34 > input[5] = 12  
    j = 2: input[2] =  7 < input[5] = 12  
       ldsLength[2] = ldsLength[2] + 1 = 2  
    j = 1: input[1] = 18 > input[5] = 12  
ldsLength[] : [2, 2, 1, 3, 2, 1, 2, 1]  
  
**lisLength[] : [1, 2, 1, 3, 3, 3, 4, 4]**  
**ldsLength[] : [2, 2, 1, 3, 2, 1, 2, 1]**  
  
Finally, length of longest bitonic subsequence is calculated:  
lisLength[1] + ldsLength[1] - 1  =  2 + 2 - 1  =  3  
lisLength[2] + ldsLength[2] - 1  =  1 + 1 - 1  =  1  
lisLength[3] + ldsLength[3] - 1  =  3 + 3 - 1  =  5  
lisLength[4] + ldsLength[4] - 1  =  3 + 2 - 1  =  4  
lisLength[5] + ldsLength[5] - 1  =  3 + 1 - 1  =  3  
lisLength[6] + ldsLength[6] - 1  =  4 + 2 - 1  =  5    ---> MAXIMUM  
lisLength[7] + ldsLength[7] - 1  =  4 + 1 - 1  =  4  
  
Hence, solution = 5  
As can be seen from any of the following bitonic subsequences.  


|  |
| --- |
| public class LongestBitonicSubsequence {        public static int longestBitonicSubsequence(int[] input) {            if(input == null || input.length == 0) {              return 0;          }            int n = input.length;          // lisLength[i] = Length of Longest Increasing Subsequence in input[0..i]          int[] lisLength = new int[n];          // ldsLength[i] = Length of Longest Increasing Subsequence in input[i..n-1]          int[] ldsLength = new int[n];          int maxLength = 1;            // Find lengths of longest increasing subsequence for all sub arrays [0..i]          lisLength[0] = 1;          for (int i = 1; i < n; i++) {              lisLength[i] = 1;              for (int j = 0; j < i; j++) {                  if (input[i] > input[j] && lisLength[i] < lisLength[j] + 1) {                      lisLength[i] = lisLength[j] + 1;                  }              }          }            // Find lengths of longest decreasing subsequence for all sub arrays [i..n-1]          ldsLength[n-1] = 1;          for (int i = n - 2; i >= 0; i--) {              ldsLength[i] = 1;              for (int j = n - 1; j > i; j--) {                  if (input[i] > input[j] && ldsLength[i] < ldsLength[j] + 1) {                      ldsLength[i] = ldsLength[j] + 1;                  }              }          }            for (int i = 1; i < n; i++) {              if (maxLength < lisLength[i] + ldsLength[i] - 1)                  maxLength = lisLength[i] + ldsLength[i] - 1;          }            return maxLength;      }        public static void main(String[] args) {          int[] array = { 12, 18, 7, 34, 30, 28, 90, 88 };          System.out.println("Length of Longest Bitonic Subsequence: " + longestBitonicSubsequence(array));      }  } |

## Order of the Algorithm

Time Complexity is O(n^2)  
Space Complexity is O(n)

# Given an array with all distinct elements, find the length of the longest sub-array which has elements(not in any particular order) that could form a contiguous sequence

Given an array with all distinct elements, find the length of the longest sub-array in that array such that the elements present in that sub-array when re-arranged could form a contiguous sequence. For example, for the array - {10,12,11,94,56,32,34,36,33,35,37}, if we consider a sub-array starting at index 0 and ending at index 2, the elements in this sub-array could be re-arranged to form a sequence 10,11,12. Similarly, for sub-array starting at index 5 and ending at index 10, if the elements in this sub-array are re-arranged we could obtain a contiguous sequence 32,33,34,35,36,37 which turns out to be longest such sub-array for this input array.  
  
Therefore, for input array {10,12,11,94,56,32,34,36,33,35,37}, output returned should be 6. If the input array is {10,12,14} then output returned should be 1. For input array {10,13,12} output returned should be 2.

## Video coming soon!

#### Subscribe for more updates

## Algorithm/Insights

The main insight used in this algorithm is that for a given sub-array if the length of the sub-array is equal to one plus difference of values of maximum and minimum element in that sub-array then the elements in that sub-array would be able to form a contiguous sequence. This condition holds true only if the elements of the sub-array are distinct elements.  
For example, for sub-array {32,34,36,33,35,37} from array {10,12,11,94,56,32,34,36,33,35,37}, maximum element in this sub-array is 37 and minimum element is 32 making the difference between them as 5. This difference of 5 plus one is equal to the length of this sub-array which is 6.  
In the algorithm, we check for the condition (maxElement - minElement + 1 = length of the sub-array) for all possible sub-arrays. If this condition evaluates to true, we update the maximum length of such sub-array if required.  
  
Time complexity for this approach is O(n^2) with O(1) extra space.

|  |
| --- |
| public class LongestSubarrayContiguousElements  {      public int findRequiredLength(int[] array)      {          if (array.length == 0)          {              return 0;          }            int maxLength = 1;          int subArrayStart = 0, subArrayEnd = 0;            // we are interested in looking at sub-arrays with minimum length as 2          // since length of 1 for longest sub-array with contiguous elements is trivial          // and we don't need to check for that          for (int i = 0;  i < array.length - 1; i++)          {              int maxElement = array[i], minElement = array[i];                for (int j = i+1; j < array.length; j++)              {                  // update min and max elements                  if (array[j] > maxElement)                  {                      maxElement = array[j];                  }                  if (array[j] < minElement)                  {                      minElement = array[j];                  }                    // if difference between min and max is equal to length of sub-array                  // then we have found sub-array with elements which could form contiguous sequence                  if ((maxElement - minElement) == (j - i))                  {                      // if length of sub-array is greater than maximum length found                      if ((j - i + 1) > maxLength)                      {                          maxLength = j - i + 1;                          subArrayStart = i;                          subArrayEnd = j;                      }                  }              }          }            System.out.println(subArrayStart + "-" + subArrayEnd);          return maxLength;      }          public static void main(String[] args)      {          LongestSubarrayContiguousElements solution = new LongestSubarrayContiguousElements();          int[] testArray = {10,12,11,94,56,32,34,36,33,35,37};            System.out.println(solution.findRequiredLength(testArray));        }  } |

## Order of the Algorithm

Time Complexity is O(n^2)  
Space Complexity is O(1)

## Contribution

# Find an integer array corresponding to the string specifying increase-decrease transitions

Given a string of size 'n' where each character can be either 'd' or 'i' and nothing else. If character 'd' denotes decrease in value and character 'i' denotes increase in value then how can we build an integer array of size 'n+1' created by using numbers from 1 to 'n+1' such that this array holds one to one correspondence with the input string.  
  
For example, if the given string is "di" then because string size is 2, we need to use numbers 1,2,3 and build integer array [3,1,2] where first pair formed by first two elements 3,1 corresponds to character 'd' since there is decrease in value from 3 to 1 and then second pair 1,2 corresponds to character 'i' - increase in value from 1 to 2. Another way to build this could have been [2,1,3]. Now pair (2,1) corresponds to 'd' and second pair (1,3) corresponds to character 'i'.  
  
When we consider 'n' such pairs formed out of adjacent elements from 'n+1' elements, 'n' pairs should correspond to 'n' characters of input string in the same sequence.  
  
Another example could be for input string 'ddddi', one of the outputs could be [6,4,3,2,1,5].  
  
Write a program to create any one of the correct output integer array given an input string having characters 'd' and 'i'.

## Algorithm/Insights

The idea is to use the higher numbers for satisfying 'i' transitions and remaining numbers for 'd' transitions.  
  
Let's look at an example. Say we have a string consisting of only 2 'i's and 3 'd's - 'ddidi'  
Then out of numbers 1,2,3,4,5,6 we reserve two higher numbers 6 and 5 for 'i' transitions because no matter what the current number we can always satisfy 'i' transitions using these higher numbers. The number next to reserved numbers for 'i' transitions is put at the 0th position. In this case, number next to 6 and 5 would be 4 hence we make output[0] = 4. Note that out of 1,2,3,4,5,6 we have marked top 2 numbers that is 6 and 5 for 'i' transitions and next number to them that is 4 as a start value for output array - therefore remaining numbers(that is 3,2,1) would be used for 'd' transitions. Now we mark lowest number from the numbers reserved for 'i' transitions as increaseValue - in this case increaseValue would be 5. Similarly, we mark highest number from numbers that are going to used for 'd' transitions as decreaseValue - in this case decreaseValue would be 3.  
  
Now the algorithm is very simple. Whenever we see a 'd' at string[i], we just put decreaseValue at output[i+1] and decrement decreaseValue by 1. And whenever we see a 'i' at string[i], we just put increaseValue at output[i+1] and increment increaseValue by 1. Output[0] is already initilaized to the number next to numbers reserved for 'i' and 'd' transitions.  
  
In this case, for string 'ddidi' we will be using numbers 1,2,3,4,5,6  
Numbers reserved for 'i' - 6,5  
StartValue to be put at output[0] - 4  
Numbers reserved for 'd' - 3,2,1  
increaseValue marked at - 5  
decreaseValue marked at - 3  
  
output[0] = 4  
output[1] = 3, decreaseValue updated to 2 (since 0th character for input is 'd')  
output[2] = 2, decreaseValue updated to 1 (first character is 'd')  
output[3] = 5, increaseValue updated to 6 (second character is 'i')  
output[4] = 1, decreaseValue updated to 0 (third character is 'd')  
output[5] = 6, increaseValue updated to 7 (fourth character is 'i')

|  |
| --- |
| public class DecreaseIncreaseSequence  {        public void createSequence(String input, int[] output)      {          if (input.length() == 0)          {              return;          }            int iCount = 0;          // count the number of increases required          for (int i = 0;  i < input.length(); i++)          {              if (input.charAt(i) == 'i')              {                  iCount += 1;              }          }            // now in numbers 1 to n+1 reserve 'iCount' higher numbers to be used for 'i'          // for example if there are 3 'i's in 6 character string,          // then reserve numbers 7,6 and 5 for 'i'            int n = input.length();            // if we see a 'i', put 'increaseValue' in the output array and increment 'increaseValue' by 1          int increaseValue = n + 2 - iCount;            // keep startValue fixed          int startValue = increaseValue - 1;            // if we see a 'd', put 'decreaseValue' in the output array and decrement 'decreaseValue' by 1          int decreaseValue = startValue - 1;            output[0] = startValue;          for (int i = 0;  i < input.length(); i++)          {              if (input.charAt(i) == 'i')              {                  output[i+1] = increaseValue;                  increaseValue += 1;              }                if (input.charAt(i) == 'd')              {                  output[i+1] = decreaseValue;                  decreaseValue -= 1;              }          }      }          public static void main(String[] args)      {          DecreaseIncreaseSequence solution = new DecreaseIncreaseSequence();            String input = "idddii";          int[] output = new int[input.length() + 1];            solution.createSequence(input, output);            System.out.print("Output sequence corresponding to input string: ");          for (int i = 0; i < output.length; i++)          {              System.out.print(output[i] + ", ");          }      }  } |

## Order of the Algorithm

Time Complexity is O(n)  
Space Complexity is O(1)

# Gold Mine Problem

Given a gold mine of n\*m dimensions. Each field in this mine contains an integer which is the amount of gold in tons. Initially the miner is in first column but can be at any row i. He can move only (right , right up , right down ) that is from a given cell, the miner can move to the cell diagonally up towards the right or right or diagonally down towards the right. Find out maximum amount of gold he can collect.

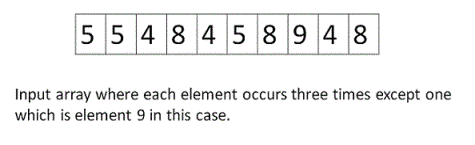
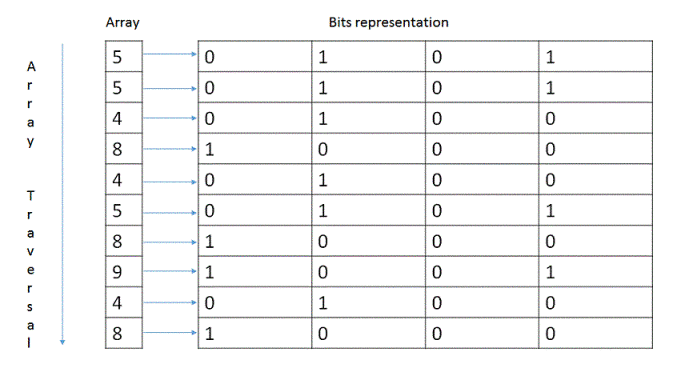
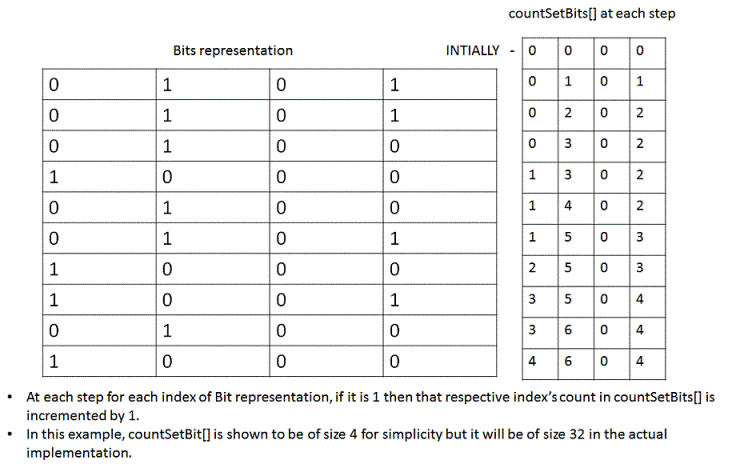
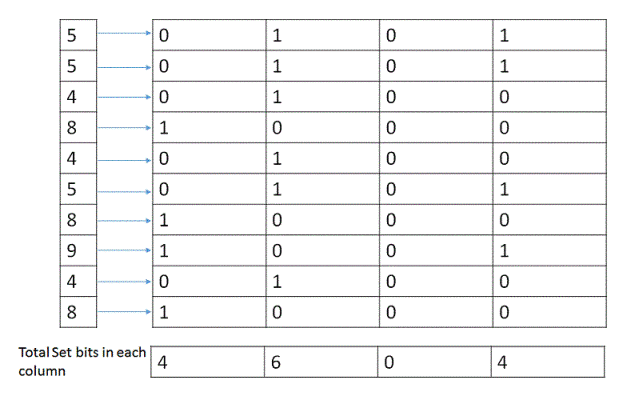
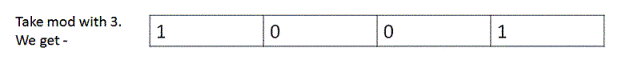
|  |
| --- |
| public class GoldMine {        private int[][] goldMine = null; // Gold mine field        public GoldMine(int[][] goldMine) {          this.goldMine = goldMine;      }        public int getMaxGold() {          // null checks          if (goldMine == null || goldMine.length == 0) {              return 0;          }          int rowLength = goldMine.length;          int colLength = goldMine[0].length;          // Create table for storing intermediate results and initialize all cells to 0          // The first row of goldMineTable will give the maximum gold that the miner can collect when he starts from that row          int[][] goldMineTable = new int[rowLength][colLength];          for (int i = 0; i < rowLength; i++) {              for (int j = 0; j < colLength; j++) {                  goldMineTable[i][j] = 0;              }          }          for (int col = colLength - 1; col >= 0; col--) {              for (int row = 0; row < rowLength; row++) {                  // Gold collected on going to the cell on the right (->)                  int right = col == colLength - 1 ? 0                          : goldMineTable[row][col + 1];                  // Gold collected on going to the cell to right up (/) i.e. diagonally up                  int rightUp = (row == 0 || col == colLength - 1 ? 0                          : goldMineTable[row - 1][col + 1]);                  // Gold collected on going to the cell to right down (\) i.e. diagonally down                  int rightDown = (row == rowLength - 1 || col == colLength - 1 ? 0                          : goldMineTable[row + 1][col + 1]);                  // Max gold collected from taking either of the above 3 paths                  goldMineTable[row][col] = goldMine[row][col]                          + Math.max(rightUp, Math.max(right, rightDown));              }          }          int max = 0;          // The max amount of gold collected will be the max value in first column of all rows          for (int i = 0; i < rowLength; i++) {              max = max < goldMineTable[i][0] ? goldMineTable[i][0] : max;          }          return max;      }        public static void main(String[] args) {            int[][] mine = { { 1, 3, 1, 5 },                           { 2, 2, 4, 1 },                           { 5, 0, 2, 3 },                           { 0, 6, 1, 2 } };            GoldMine goldMine = new GoldMine(mine);          int maxGold = goldMine.getMaxGold();          System.out.println(maxGold);      }    } |

## Order of the Algorithm

Time Complexity is O(m\*n)  
Space Complexity is O(m\*n)

# Find the Element That Appears Once in an Array

## Algorithm/Insights

The idea is to count the set bits in all the given numbers. As we know, all the elements(except one) occur 3 times so their respective set bits will also occur 3 times. Thus when we take modulus 3 of the sum of all set bits we will be left with only the bits of that number which occurs once.  
  
The steps of the algorithm are as following:   
1: Create countSetBits[] array of size 32(for representing 32 bit integer) where,  
   countSetBits[i] represents count of ith set bit of all elements in the input array.  
   Initially all elements of countSetBits[] array are 0.  
2. Traverse all the elements of the input array to populate countSetBits, by doing step #3 for each of them.  
3. Take the element and check for its set bits. If the ith bit is found to be set, then in the countSetBits[] array increment the count of the element at the index 'i'.  
  
  
  
4. After finishing the above operation for all the elements of the input array, the elements of countSetBits[] would represent count of all set bits in the elements of input array. Perform the modulus 3 operation on each element of the countSetBits[] array. Taking mod with 3 will determine the number of times that respective index was set in all the elements of given input array. If we get a remainder 1 at an index 'j', then that means in the number that occurs only once, we have a set bit of index 'j'. We will get a remainder 2 only if the given question is violated with two instances of a number.  
  


## Finally after modulus of each element in countSetBits[], simply transform the binary representation into decimal.

# Find median of two sorted arrays

Given two sorted arrays a and b each of size 'n', find the median of the array obtained by merging these two arrays.   
For example - for following two arrays 'a' and b',     
a = 1, 3, 5, 11, 17  b = 9, 10, 11, 13, 14   Sorted(a+b) = 1, 3, 5, 9, 10, 11, 11, 13, 14, 17 and therefore Median = (10 + 11)/2 => 10.5  
output returned should be 10.5.

## Algorithm/Insights

If array 'a' and 'b' both are of size 'n' and are sorted then to find out the median of new array obtained by merging 'a' and 'b', following algorithm is used -   
1. Base Case - If size of both arrays is 2 then use this formula to get the median. Median = (max(a[0], b[0]) + min(a[1], b[1]))/2   
    
Recursive Approach:  
2. Calculate the medians for the input arrays 'a' and 'b' respectively. Say 'm1' is median of array 'a' and 'm2' is median of array 'b'.  
3. If value of m1 is equal to m2 then we don't have to compute any further. Return m1 or m2.  
4. If m1 > m2, then the median of the merged array must be present in one of the following two sub-arrays -  
    a) Subarray starting from element m2(including m2) and ending at the last element of array 'b'. Therefore, we modify low index of array 'b' to the index of element m2.  
    b) Subarray starting from the first element of array 'a' and ending at element m1(including m1). Therefore, we modify high index of array 'a' to the index of element m1.  
5. If m2 > m1, then the median of the merged array must be present in one of the following two sub-arrays -  
   a) Subarray starting from element m1(including m1) and ending at the last element of array 'a'. Therefore, we modify low index of array 'a' to the index of element m1.  
   b) Subarray starting from the first element of array 'b' and ending at element m2(including m2). Therefore, we modify high index of array 'b' to the index of element m2.  
6. With these modified low and high indices, we jump to step #2 until we hit base case(step #1).

# Find Majority Element in an Array

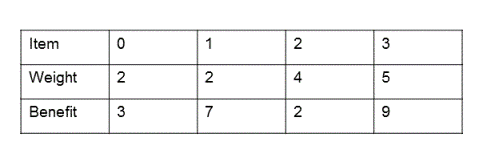
Given an array of size n, find the element which occurs more than n/2 times. This element is called Majority Element.

|  |
| --- |
| public class MajorityElement {        // Boyer-Moore Vote Algorithm      public static Integer getMajorityElement(int[] array) {            if (array == null || array.length == 0) {              return null;          }            // Step 1: Find max element          Integer candidate = null;          int count = 0;          for (int i = 0; i < array.length; i++) {              if (count == 0) {                  candidate = array[i];                  count = 1;                  continue;              } else {                  if (candidate == array[i]) {                      count++;                  } else {                      count--;                  }              }          }            if (count == 0) {              return null;          }            // Step 2: Check if candidate is majority element          count = 0;          for (int i = 0; i < array.length; i++) {              if (candidate == array[i]) {                  count++;              }          }          return (count > array.length / 2) ? candidate : null;      }        // Naive Algorithm      public static Integer getMajorityElementNaive(int[] array) {            if (array == null || array.length == 0) {              return null;          }            for (int i = 0; i < array.length; i++) {              int count = 0;              for (int j = 0; j < array.length; j++) {                  if (array[i] == array[j]) {                      count++;                  }              }              if (count > array.length / 2) {                  return array[i];              }          }          return null;      }        public static void main(String[] args) {            int[] array = { 2, 6, 2, 2, 6, 2, 2, 8, 2, 1 };          System.out.println(Arrays.toString(array) + " \nMajority Element: " + getMajorityElement(array));        }  } |

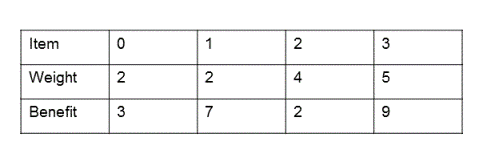
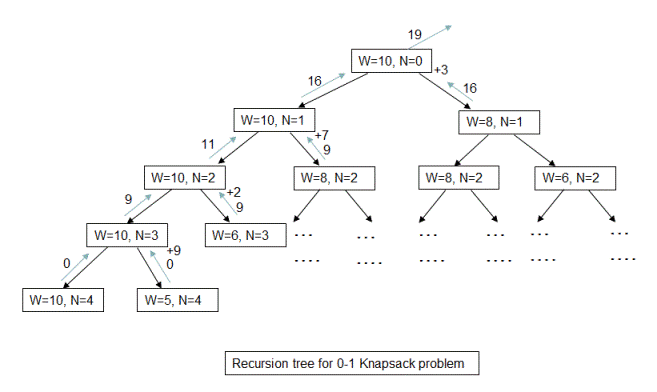
## Order of the Algorithm

Time Complexity is O(n)  
Space Complexity is O(1)

# 0-1 Knapsack Problem

Given a set of items, each with weight and benefit, determine the items to include in the knapsack so that the total weight is less than or equal to a given weight limit and the total benefit is maximized. For example, if we are given following four items with their corresponding weights and benefits which items should we include in the knapsack to maximize the benefit. The weight limit for this knapsack is 10?  
  
As you can verify, the items to include in the knapsack(with the weight limit of 10) to get the maximum benefit are item #1, item #2 and item #4. Maximum benefit obtained in that case is 19 and the weight of the knapsack becomes 9 which is within the given weight limit.

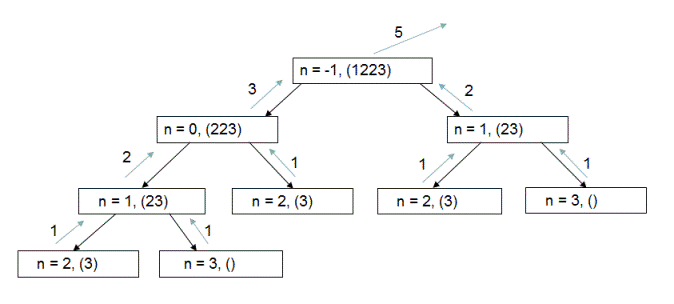
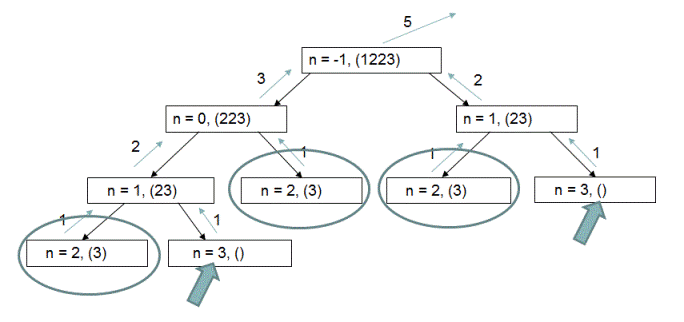
## Algorithm/Insights

Given an item 'n', with benefit v1 and weight w1, we basically have only two choices: 1. To include the item in the knapsack 2. To exclude the item from the knapsack. At each item 'n', we compute the benefit that could be obtained in both include and exclude cases and choose the case which gives more benefit. If we choose to include the item 'n', then we decrease the weight limit by w1 and then compute the maximum possible benefit using items from 'n+1' onwards(benefit\_n\_Plus). This maximum possible benefit(benefit\_n\_Plus) + (benefit of including item 'n') is maximum benefit obtained in the include case for item n. Now in the exclude case, weight limit is not modified and maximum possible benefit is computed from item 'n+1' onwards. This computed benefit is the maximum possible benefit for item 'n' in exclude case. We compare the benefits obtained in include and exclude case and return the maximum of these two values.    
  
Note that to compute maximum possible benefits from item 'n+1' onwards, recursive calls are made with modified weight limit in the include case. Base case for this recursion would be when all the items are considered or when the weight limit remaining is 0. In both of these cases, benefit returned is 0.  
  
For the following example  
  
partial recursion tree depicting the include and exclude cases is shown below. At each state, 'W' denotes the weight limit available, 'N' denotes the index of the item in the input item array. Left branches depict the exclude case where item 'N' is not included and hence weight limit is not modified. Right branches depict the include case where weight limit is modified and also benefit of including that item is added to returned benefit. In the recursion tree, (W=10,N=4) and (W=5,N=4) states are base cases where there are no more items that could be included and hence these state return benefit of 0.  
  
Please checkout function findMaximumBenefit(int w, int n, int [] val, int [] weight) in code snippet which implements this recursive algorithm.  
  
**Dynamic programming/Memoization:**The time complexity of above algorithm is exponential. Now in this recursive algorithm, many of the states like (W=8,N=2) shown in recursion tree are repeated and hence redundant computations are performed. We use top-down dynamic programming approach which stores the solution of intermediate sub-problems and re-uses them if required. Please checkout function findOptimalItems(int w, int n, int [] val, int [] weight, ListWithBenefit[][] optimalKnapsack) in code snippet for implementation details. Time complexity of this approach is O(wn) where w is weight limit and n is total number of items. Space complexity is also O(wn).  
  
Formal steps of above algorithm are as following. Along with the maximum possible benefit, it also returns the items that need to be included in the knapsack for maximum benefit.    
  
1. Base case:   
a. If weight limit is 0 then no benefit can be obtained and hence we return 0 as benefit with an empty collection.  
b. If the current item number that we are trying to include is greater than number of items given to us then this is invalid case. And we return 0 as benefit with an empty collection.  
                 
Recursive steps:  
2. If weight of the current item is greater than the weight limit we have then we do not include this item in knapsack and look for potential items to include from next item onwards.  
3. Otherwise,   
a. we check the benefit that could be obtained by including current item (say benefit = v1) and optimal benefit from remaining items(say benefit = v2). Therefore, total benefit that could be obtained would be (v1 + v2) say 'benefit\_include'.  
b. we also check the optimal benefit that could be obtained by excluding current item and obtaining optimal benefit from remaining items say 'benefit\_exclude'.  
4. We return the maximum of 'benefit\_include' and 'benefit\_exclude' with the corresponding collection.

# Count all possible decodings of a given digit sequence

Let's say there is an encoding scheme where integer 1 encodes character 'A', integer 2 encodes character 'B' and so on till integer 26 which encodes character 'Z'. With this encoding scheme, write a program to count all possible decodings of a given digit sequence.  
  
For example, if the digit sequence is "12" then there are two possible decodings for this  -  One of them is 'AB' when we decode 1 as 'A' and 2 as 'B'. Now we can also decode this digit sequence "12" as 'L' when we decode digits 1 and 2 taken together as an integer 12.  
Similarly, for digit sequence "1223" there are five possible decodings.  
1, 2, 2, 3 - ABBC  
1, 2, 23   - ABW  
1, 22, 3   - AVC  
12, 2, 3   - LBC  
12, 23     - LW  
  
For digit sequence "1234" there are three possible decodings.  
1, 2, 3, 4 - ABCD  
1, 23, 4   - AWD  
12, 3, 4   - LBD  
Note that because integer 34 does not have any valid decoding, we cannot decode "1234" as 1,2,34 or as 12,34 where digits 3 and 4 are decoded together(as an integer 34) to some character.  
  
Few assumptions that we can make here -   
1. Number of possible decodings for an empty sequence = 0.  
2. There would be no 0's at the very beginning of digit sequence. That means, no digit sequence would start from digit 0.  
3. There would be no two or more consecutive 0's in the digit sequence.    
4. The digit sequence would be formed by only using digits from 0-9.

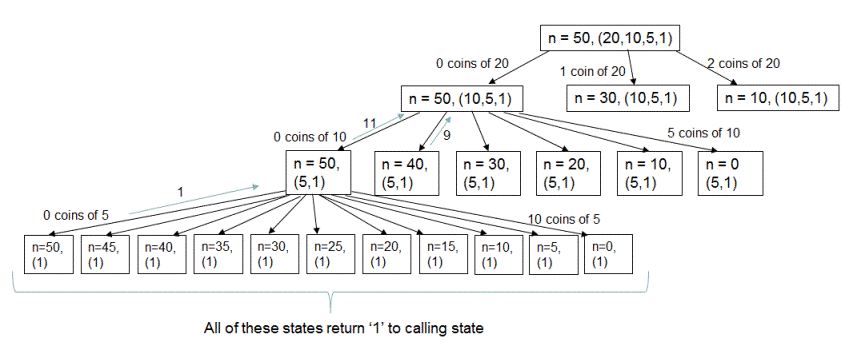
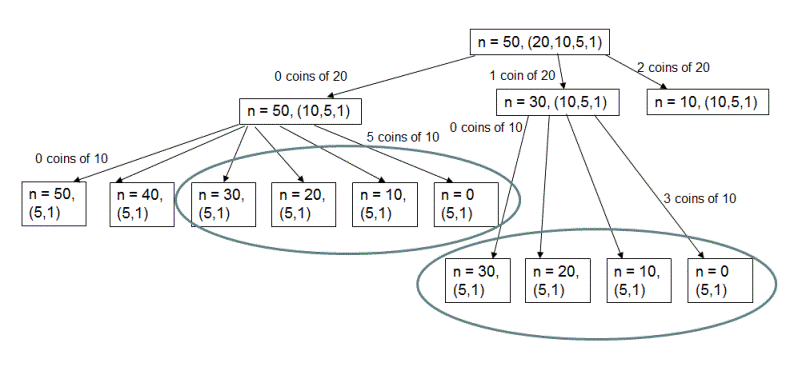
## Algorithm/Insights

The algorithm to solve this problem is recursive in nature. Let the function that implements this algorithm be 'countPossibleDecodings(int n, int[] digitSequence)' where 'n' denotes the number of digits already included from digitSequence while counting number of possible decodings. Therefore, in 'countPossibleDecodings(int n, int[] digitSequence)' we are essentially counting number of decodings possible for digit sequence starting at index 'n+1' and ending at index (digitSequence.length-1) in digitSequence array. Therefore the initial call to this function is made with n = -1 (countPossibleDecodings(-1, digitSequence)).  
  
Now in the function countPossibleDecodings(int n, int[] digitSequence)  
1. If 'n' is equal to (digitSequence.length-2) then that means there is only one digit to count possible decodings for. In this case only 1 decoding is possible and hence we return 1.  
2. If 'n' is equal to (digitSequence.length-1) then that means there are no digits to count possible decodings for and hence we return 1 as per the assumption - number of decodings for an empty sequence = 1.  
  
Step #1 and #2 are base cases for this recursive algorithm.  
3. Now remember that in countPossibleDecodings(int n, int[] digitSequence), we are computing total number of decodings for digitSequence starting at index 'n+1' and ending at the end of the sequence. If digit at index 'n+1' is greater than 0 then that digit could be decoded and appended with all possible decodings of digitSequence starting at index 'n+2' and ending at the end of the sequence. Therefore total number of decodings possible by decoding digit at index 'n+1' are counted by making a recursive call countPossibleDecodings(n+1, digitSequence).  
4. In the call countPossibleDecodings(int n, int[] digitSequence), in above step we chose to decode digit at index 'n+1' and append it with all possible decodings for digitSequence from index 'n+2' to end of the sequence. We could have also chosen to decode two digits taken at once that is digit at index 'n+1' and digit at index 'n+2' if they form a valid two digit integer which is greater than 10 and less than 27(only in this case we get different and valid decodings). To count the number of such decodings possible, we check if integer formed using digits at indices 'n+1' and 'n+2' is a valid integer and then make a recursive call countPossibleDecodings(n+2, digitSequence).  
  
Please checkout following function call tree for digitSequence "1223" which shows how the recursive calls are made starting from n=-1 and what are the values returned from each function call.  
  
The time complexity for this algorithm is O(2^n) as can be easily seen from the number of function calls in above function call tree. Please checkout function countPossibleDecodings(int n, int[] digitSequence) for implementation details.  
  
  
Now as you can observe in the above function call tree, we are doing redundant computations for n=1, n=2 and n=3. To avoid these redundant computations, we store the computed values in an array and use these stored values if needed at a later stage of the algorithm. With this optimization, we need to compute all possible decodings only once for each value of 'n'. Time complexity therefore is reduced to O(n) with extra space usage of O(n). Please checkout function countPossibleDecodings(int n, int[] digitSequence, int[] decodings) for implementation details.

# Find total number of ways to make change using given set of coins

Given an infinite supply of coins of denominations {20,10,5,1}, find out total number of way to make change of given amount - 'n'?   
For example, if given amount is 20, there are 10 ways to make this change as shown below -   
(1 of 20),(1 of 10 + 2 of 5),(1 of 10 + 1 of 5 + 5 of 1),(1 of 10 + 10 of 1), (2 of 10), (1 of 5 + 15 of 1),(2 of 5 + 10 of 1),(3 of 5 + 5 of 1),(4 of 5),(20 of 1)  
  
If the amount given is 0 then the total number of ways to make change is 1 - using 0 coins of every given denomination.

## Algorithm/Insights

Let's try to understand this algorithm using an example. If we are make change of 50 using infinite number of coins of denominations {20,10,5,1} then   
  
total number of ways to make change of 50 using denominations {20,10,5,1} = total number of ways to make change of 50 using 0 coins of 20 + total number of ways to make change of 50 using 1 coin of 20 + total number of ways to make change of 50 using 2 coins of 20  
  
Now first term on the right hand side of above equation that is - total number of ways to make change of 50 using 0 coins of 20 can be restated as total number of ways to make change of 50 using denominations {10,5,1}  
  
And second term that is total number of ways to make change of 50 using 1 coin of 20 = total number of ways to make change of 30 using denominations {10,5,1}  
  
Similarly, total number of ways to make change of 50 using 2 coins of 20 = total number of ways to make change of 10 using denominations {10,5,1}.  
  
As you can see, this algorithm is recursive in nature and the recursion tree for the above example looks like following. Only one complete path is shown in recursion tree due to space constraint.  
  
The base case for this algorithm would be when the denomination set has only coins of 1 in it. In that case total number of ways to make change would be 1. Also when amount to make change of is 0, total number of ways to make change would be 1(0 coins of all denominations).  
  
The formal steps of this algorithm are -   
1. If the current denomination is 1 then return 1. This is the base case.  
2. If current denomination is 20, set next denomination as 10; if current denomination is 10, set next denomination as 5 and if current denomination is 5, set next denomination as 1.  
3. Now implement the recurrence relation: numberOfWays(amount, denom) =  numberOfWays(amount - 0\*denom, nextDenom) + numberOfWays(amount - 1\*denom, nextDenom) + ... + numberOfWays(0, nextDenom) using a while loop.  
  
The time complexity of this algorithm is exponential as can be easily observed from recursion tree.  
  
**Dynamic Programming - Memoization approach**: For the same example, if we look at the recursion tree shown below which highlights the re-computations for the sub-problems of n = 30 and  denominations = {5,1}, n = 20 and  denominations = {5,1} and so on.  
  
To avoid these re-computations, we could store the results when computed and re-use them if required again. This reduces the time complexity of this algorithm to O(nm) where n is total amount to make change for and m is total number of denominations. For the example shown in the recursion tree n would be 50 and m would be 4. This approach takes extra space of O(nm). Please checkout function countNumberOfWays(int amount, int denom, HashMap numberOfWays) from code snippet for implementation details.

# Find increasing sub-sequence of length three having maximum product

Given an array of positive numbers, find sub-sequence of length three having maximum product. The elements of sub-sequence should be in increasing order. For example, for the input array {6, 7, 8, 1, 2, 3, 9, 10} output should be sub-sequence 8,9,10. For input array {6, 1, 2, 3, 19, 10, 7} output sub-sequence should be 2,3,19.

## Algorithm/Insights

The idea is - for given array 'a' and for every element a[i], find out the largest element-'leftLargest' which is less than a[i] in left sub-array from index 0 to index 'i-1', and find out the largest element-'rightLargest' which is greater than a[i] in right sub-array from index 'i+1' to 'size of array'. The product leftLargest\*a[i]\*rightLargest would be the largest possible product of increasing sub-sequence of three elements which includes a[i]. If for any element a[i], leftLargest or rightLargest value is not found then we set that to 0 since no triplet is possible by including this element 'a[i]' in mid-position of sequence. We find out this product(leftLargest\*a[i]\*rightLargest) for all 'a[i]' values and find out the maximum value from these values.      
  
You may want to checkout function - findSequence(int[] array, int[] sequenceIndices) for implementation details.

# Find increasing sub-sequence of length three having maximum product | Optimized approach

Given an array of positive numbers, find sub-sequence of length three having maximum product. The elements of sub-sequence should be in increasing order. For example, for input array {6, 1, 2, 3, 19, 10, 7} output sub-sequence should be 2,3,19. For the input array {6, 7, 8, 1, 2, 3, 9, 10} output should be sub-sequence 8,9,10.

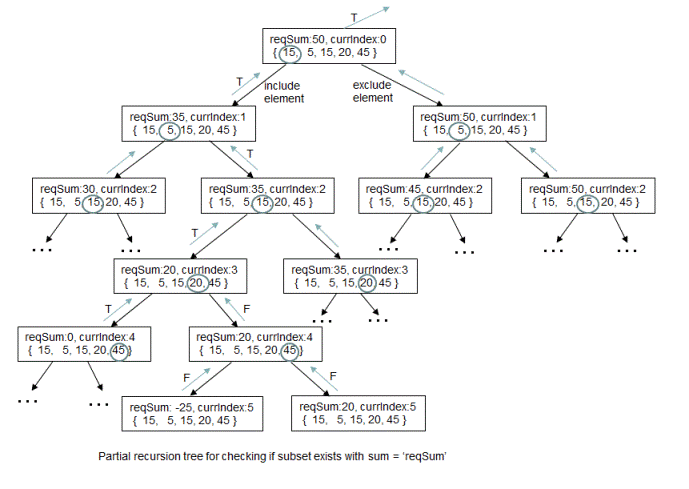
## Algorithm/Insights

For every element a[i] in array 'a', we need to find the largest element less than a[i](say leftLargest[i]) in left sub-array a[0 .. i-1] and largest element greater than a[i](say rightLargest[i]) in right sub-array a[i+1 .. lengthOfArray-1]. Once we have leftLargest[i] and rightLargest[i], we know that no increasing sequence of length 3 with a[i] as the second element in it would yield greater value for product than sequence leftLargest[i],a[i],rightLargest[i]. We compute this product leftLargest[i]\*a[i]\*rightLargest[i] for all elements in array 'a' and pick the maximum of these products to return the sequence which produces this maximum product.  
  
Now let's see how can we compute leftLargest[i] and rightLargest[i] for all elements in array 'a'.  
1. To compute leftLargest[i] for an element a[i], we make use of AVL tree to compute floor for each element. Floor of an element 'n' for a given array is the largest value in the given array which is less than element 'n'. After computing floor of an element a[i], a[i] is inserted into the AVL tree. If floor of a[i] is not found then leftLargest[i] is made 0. This step takes O(nlogn) time.  
  
We have discussed how to calculate floor of an element in [this post.](http://www.ideserve.co.in/learn/floor-ceiling-using-binary-search-tree) Please note that floor definition here is slightly different than in the post because here we are looking for sequence which is strictly increasing. Checkout function getFloor() in the code snippet below which implements this functionality.   
  
2. To compute rightLargest[i] for an element a[i], we start traversing the array from rightmost end(from index arrayLength-1). While traversing the array, we keep track of the largest element seen so far(say 'rightLargestSofar') in sub-array a[i+1 .. (arrayLength-1)]. If the value 'rightLargestSofar' is greater than a[i] then rightLargest[i] is assigned a value 'rightLargestSofar' otherwise it's assigned the value as 0; also 'rightLargestSofar' is updated to a[i]. In this way, complete array rightLargest is computed. This step takes O(n) time. Checkout function fillRightGreatest(int[] array, int[] rightGreatest) for implementation details.  
    
3. After computing leftLargest[i], rightLargest[i] for all a[i]'s , we pick the maximum of products leftLargest[i]\*a[i]\*rightLargest[i] to output the required sequence. This step takes O(n) time. Please checkout function - findSequence(int[] array, int[] sequenceIndices) for implementation details.  
  
Overall time complexity of this algorithm is O(nlogn) which comes from step #1 while computing 'leftLargest' array. Extra space required is O(n).

# Set Partition Problem | Recursion

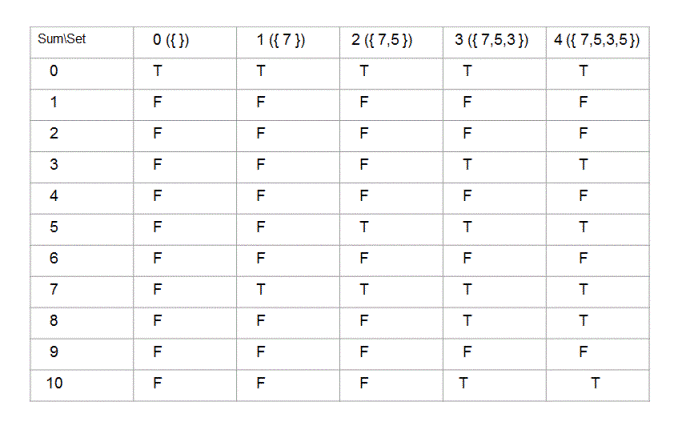
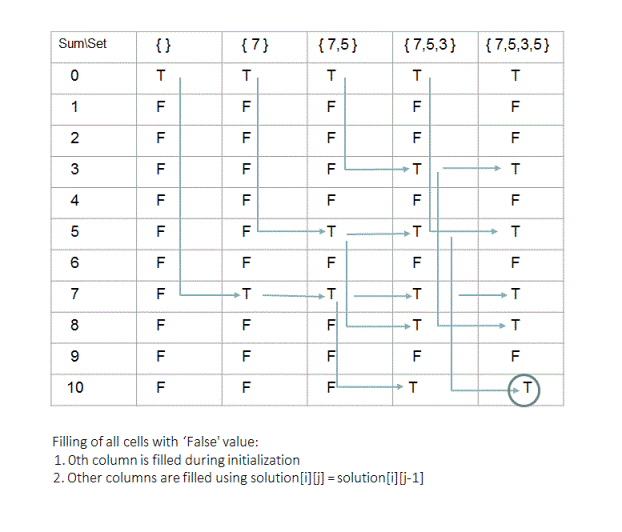
Given a set, find out if it can be partitioned into two disjoint subsets such that sum of the elements in both these subsets is equal. Intersection of disjoint sets should be null and union should be the complete set.  
For set {15,5,15,20,45}, sets {15,5,15} and {20,45} are valid disjoint sets. Also sets {15,5,45} and {15, 20} are valid disjoint sets. But sets {15,5,45,20} and {20,45} are not valid disjoint sets since element 20 is present in both of these subsets.   
  
Coming back to the problem of finding two disjoint sets with equal sums, an example for that could be subsets {15,15,20} and {5,45} for set {15,5,15,20,45}. Sum of elements in both of these subsets is 50. Therefore, the output returned by the program should be 'True'.  
  
Output returned by the program should be 'false' for the set {21,33,37,2}. You can verify that, this set cannot be partitioned into any valid combination of two disjoint subsets such that sum of elements in these subsets is equal.

## Algorithm/Insights

First simple idea that we are going to use here is that if sum of all elements in the given set is an odd number say '2n+1', then the best we might be able to do is to partition the given set into two subsets - one with sum 'n' and another with sum 'n+1'. In other words, if sum of all the elements in given set is odd, there is no way the set can be partitioned into two subsets with equal sum and therefore we return 'false' in this case..   
  
Now if the sum of all elements in the given set is an even number say '2n', then all we need to do is to find out a subset say 'S1' of the given set such that sum of elements in that subset is 'n'. The remaining subset(obtained by excluding  elements in 'S1') then is guaranteed to have sum = 'n'. For example, the sum of all elements for this set {15,5,15,20,45} is 100. And the subset {5,45} has sum = 50. Now the remaining subset obtained by excluding elements 5 and 45 is {15,15,20} which also has sum = 50.  
  
Notice how we have reduced the original problem into the problem of finding a subset with sum = 'n' when sum of all elements in given set is '2n'. Let's try to solve this modified problem now.  
  
For each element in the set say 'current element', we test for two conditions:   
1. If the subset with sum = 'n' could be obtained by including the 'current element'.   
2. If the subset with sum = 'n' could be obtained by excluding the 'current element'.   
If any of these two conditions is true, we return true.  
  
In the first case, since we are trying to find the subset with sum = 'n' by including the 'current element', we reduce the sum 'n' by value of 'current element' and find out if we can get a subset with sum = 'n - value of current element' from remaining elements of the set.  
  
In the second case, since we are excluding the 'current element' from subset, we try to find out the subset with sum = 'n' from remaining elements of the set.  
  
These two conditions can be tested using recursive calls. If the function call - 'partitionPossible(int requiredSum, int currIndex, int[] set)' is made to check if there is any subset with sum = 'requiredSum' in set(int[] set) with elements starting from 'currIndex' then to test condition #1 we call 'partitionPossible(requiredSum - set[currIndex], currIndex+1, set)' and to test condition #2 we call 'partitionPossible(requiredSum, currIndex+1, set)'.  
  
The base cases for this recursion would be -   
1.If the requiredSum is '0' then return 'true'. This is because we can always find an empty subset which would have sum = 0.   
2.If requiredSum is not '0' and there are no more elements to look at in the set, that is 'currIndex = set.length' then return 'false'.  
  
For finding out if there is any subset with sum = 50 in the set {15,5,15,20,45}, generated recursion tree using function calls 'partitionPossible' is shown below. The recursion tree is partial recursion tree. Including element at 'currIndex' takes the current state to its left child state and excluding element at 'currIndex' takes it to its right child state.  
  
  
You can observe in the recursion tree how including elements at currIndex = 0, 2 and 3 results in subset with sum = 50.  
This path would be left-right-left-left branches from the starting state.  
  
As can be easily observed, the time complexity of this algorithm is O(2^n) and space complexity is O(1). For implementation details of this algorithm, check out function 'partitionPossible(int requiredSum, int currIndex, int[] set)' in code snippet. Please add comments below in case you have any feedback/queries.

# Set Partition Problem | Dynamic Programming

In one of the previous posts, we have looked at the[recursive solution](http://www.ideserve.co.in/learn/set-partition-problem-recursion)for set partitioning problem. In this post, we will cover the dynamic programming approach to solve the same problem. If you have already read the previous post with recursive solution, you can directly skip to 'Algorithm/Insights' section.  
  
Given a set, find out if it can be partitioned into two disjoint subsets such that sum of the elements in both these subsets is equal. Intersection of disjoint sets should be null and union should be the complete set.  
For set {7,5,3,5}, sets {7,5} and {3,5} are valid disjoint sets. Also sets {7,5,3,5} and {} are valid disjoint sets. But sets {7,5,3} and {3,5} are not valid disjoint sets since element 3 is present in both of these subsets.   
  
Coming back to the problem of finding two disjoint sets with equal sums, an example for that could be subsets {7,3} and {5,5} for set {7,5,3,5}. Sum of elements in both of these subsets is 10. Therefore, the output returned by the program should be 'True'.  
  
Output returned by the program should be 'false' for the set {21,33,37,2}. You can verify that, this set cannot be partitioned into any valid combination of two disjoint subsets such that sum of elements in these subsets is equal.

If sum of all the elements in the given set is an odd number say '2n+1', then the best we might be able to do is to partition the given set into two subsets - one with sum 'n' and another with sum 'n+1'. In other words, if sum of all the elements in given set is odd, there is no way the set can be partitioned into two subsets with equal sum and therefore we return 'false' in this case.   
  
Now if the sum of all the elements in the given set is an even number say '2n', then all we need to do is to find out a subset say 'S1' of the given set such that sum of elements in that subset is 'n'. The remaining subset(obtained by excluding  elements in 'S1') then is guaranteed to have sum = 'n'. For example, the sum of all elements for this set {15,5,15,20,45} is 100. And the subset {5,45} has sum = 50. Now the remaining subset obtained by excluding elements 5 and 45 is {15,15,20} which also has sum = 50.  
  
Notice how we have reduced the original problem into the problem of finding a subset with sum = 'n' when sum of all elements in given set is '2n'. Let's try to solve this modified problem now.  
  
We are going to use dynamic programming(tabulation) approach to solve this problem. We will fill up the 'solution' array shown below in bottom up manner.  
  
  
Here, value of solution[i][j] represents if sum of 'i' could be obtained by any subset of the set having first 'j' elements (from the given set) in it. For example, for the set {7,5,3,5} solution[8][3] would be 'true' since sum of 8 could be obtained by a subset({5,3}) of the set {7,5,3}(set obtained by considering first 3 elements of the given set).  
  
Here is how we fill up this solution array -   
**Initialization:**   
1. All elements in 0th row are initialized to 'true' because for any given set, we can always find an empty subset with sum = '0'  
  
2. All elements in 0th column except the element in the 0th row are initialized to 'false' since no sum except 0 could be obtained by any subset of the empty set({}).  
  
  
**Rules for filling up solution[i][j]:**  
1. If solution[i][j-1] is 'true' then solution[i][j] is also 'true'. This is because if there exists a subset(with sum = 'i') of set formed by first 'j-1' elements(of given set) then that same subset will also be a subset for the set formed by first 'j' elements.  
  
2. If solution[i][j-1] is 'true' then solution[(i + set[j-1])][j] is also 'true'. This is because if there exists a subset(with sum = 'i') of set formed by first 'j-1' elements(of given set) then inserting 'j'th element into that subset results in a new subset which will have sum = 'i' + value of 'j'th element. This sum is represented by the row = 'i + set[j-1]'. Remember indexing scheme used here is 0 based and therefore value of 'j'th element is represented by set[j-1].  
  
3. If solution[i][j] is not set to 'true' using above two rules then set solution[i][j] to 'false'. Because there is no other case other than above two cases in which solution[i][j] would be 'true'.  
  
Using above two rules and initialization, function 'partitionPossible(int requiredSum, int[] set)' fills up 'solution' array in bottom up manner as shown below.  
  
  
**Return value:**Note that if we are looking for a subset of the given set with sum = 'requiredSum' then number of rows(numRows) for above 'solution' array would be 'requiredSum + 1' and number of columns(numCols) would be 'size of given set + 1'. Once we fill up this 'solution' array, we need to return the value of solution[numRows-1][numCols-1].  
  
Please check out the code snippet for implementation details. The time complexity of this approach is O(nm) where size of the given set is 'm' and sum of all elements in the given set is '2n'. The auxillary space required is also O(nm).